POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master's Degree Thesis

Startup performance analysis and optimization of an Android banking application

Supervisor
prof. Fulvio CORNO

Candidate
Dario Piazza

Company Tutor
Alessandro ROTA

December 2021

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Mobile banking apps are applications that are developed expressly for a certain bank, to allow the customers to access most of the services provided by the bank using a mobile device, like a smartphone or a tablet. Also considering the increasing usage of such devices, these kinds of applications are becoming more and more diffused. In fact, a study conducted by DataProt in 2020[1] has shown that "86.5% of Americans used a mobile device to check their bank balance in 2020".

However, to guarantee a high level of security to the users, the implementation logic of these applications can become very complex, and the large number of instructions that need to be executed can compromise the performance of the application itself.

For this reason, it is necessary to identify the critical sections in terms of time required to be completed, discuss possible improvements, and propose a solution to adopt them, while keeping the functionalities of the application intact

1.1 Context of the thesis

This work focuses on the analysis of the performance at launch time of a famous mobile banking Android application (it will be referred to as **Mobile Banking**), with more than 5.000.000 downloads on the Play Store. The app allows the customers to execute several operations related to banking account management like accessing their bank account, executing wire transfers and

[1] Mobile Banking Statistics That Show Wallets Are a Thing of the Past

managing users' credit cards, directly from their smartphone.

1.1.1 State of the art

The choice of focusing on the application launch phase derives from a study of the company of the **Mobile Banking** application, that highlights long startup times from users' activity. Since the app is so popular, there is a great variety in the devices on which the application is installed, both in terms of *recentness* and also hardware features. In table 1.1 it is possible to observe the distribution of the Android OS API level for the devices using the **Mobile Banking** application, as well as the average launch time registered in the corresponding devices (data are taken from the **Firebase Performance monitoring** console, whose details will be presented in section 3.4):

OS API level	Release date[12]	% of devices	Avg. launch time
29	03/09/2019	34.84%	2.99s
30	03/09/2020	33.64%	2.26s
28	06/08/2018	16.39%	4.49s
26	21/08/2017	6.7%	5.26s
24	22/08/2016	2.68%	7.34s
27	05/12/2017	2.51%	5.91s
23	02/10/2015	1.55%	14.98s

Table 1.1. Table listing the distribution of OS API level for Android devices using the **Mobile Banking** app and the corresponding average launch time. Versions with a diffusion <1% have not been reported,

Even though the release date of the OS version does not necessarily implies that the device is older (in this case its system may have been updated subsequently) or newer (the device was released after the OS release date) the table gives an idea of the different devices that are used. However, the performance of devices running more recent OS API levels are not always better than older ones, as the hardware support of the smartphone is determinant. Unfortunately, the details about the hardware characteristics of the devices are not available.

In table 1.2, instead, are reported the percentile information about the app startup time, as retrieved from the **Firebase Performance Monitoring** platform:

Percentile	Startup time
50 % (median)	3.11s
75%	4.67s
85%	5.76s
90%	6.86s
95%	9.45s

Table 1.2. **Mobile Banking** app startup time percentile distribution.

It is possible to notice that the high number and variety of devices running the application is also reflected on the time needed to launch the application in a **COLD** state (meaning that no information about the app are already present in memory, so this is the worst-case scenario, for details see section 2.1.3). Even though the way in which the **Firebase Performance Monitoring** tool registers the startup time of the app is slightly different from the one that will be used throughout this work[2], these metrics allows us to understand that a reasonable startup time (below 5s, according to **Android Vitals**, see section 3.5 documentation for details) is registered for approximately 80% of devices, while the remaining 20% of users is likely experiencing what is defined a **slow COLD start**.

The **Mobile Banking** company recognized that a slow startup can reduce customers' satisfaction and negatively impact their business. From here, the idea of carrying out a deeper analysis of the performance of the application at launch time.

1.2 Goal of the thesis

The goal of the thesis is to employ the tools available for the performance analysis of Android applications to identify possible issues in the **Mobile Banking** app, that negatively affect its launch time. For most of the problems discovered through this analysis, a potential solution will be presented and implemented. The final objective of this work is to show that, by following the correct guidelines and solving the performance problems found by utilizing the tools, the time required to launch the application from scratch

[2]for details on Firebase Performance Monitoring start time refer to this link, while the one we will be considering is presented in section 2.1.

can be significantly reduced, considering not only high-end devices, but also less performing ones.

1.3 Structure of the work

In the first sections of this document, it will be presented an overview of the operations that are performed by the Android operating system when launching an Android application. After this introduction, the thesis will illustrate which are the most common factors that can influence the startup of an Android application and the tools that are available to analyze the performance of the app.

Subsequently, this work will consider the problems presented in the aforementioned sections in the context of the **Mobile Banking** application, highlighting how they are identified and the impact they have on the performance at launch time. For the critical points discovered through the analysis a solution will be proposed and, whenever possible, implemented. In this case it will also be shown how the suggested approach influence the startup time of the application.

The final sections will be dedicated to the comparison of the application launch time between the initial state of the software and the version including the proposed modifications, considering different kinds of devices. The results obtained will be analyzed and discussed in the final chapter.

1.4 Importance of application startup

One of the most important aspects of a mobile application is its responsiveness and, in particular, how fast it is in allowing users to perform the desired operations. For this reason, starting from the first usage, users expect the application to be launched quickly, otherwise, the experience can be frustrating and it can bring them to uninstall the app right away or publish a negative review on the Play Store. In fact, a survey conducted by Statista Research Department from 2010 to 2019, showed that the mobile application abandonment rate after the first usage is nearly 25%[2]. This is also due to the fact that, nowadays, the number of applications that are released every month on the App stores is in the order of hundreds of thousand [3], and the

[3] Statista Research Department study - 2021

competition among companies that offer similar products is fierce.

Therefore, it is likely that users that give up on using a certain application will try to switch to a similar one that offers comparable functionalities, but better performance.

On the other hand, applications whose performance satisfies users' expectations play a major role in brand promotion and recognition. In fact, Google's analysis on Play Store reviews has shown that "when leaving a 5-stars review, 73% of users mention speed, design, and usability" [10]. In addition to that, a 2019 survey from Apptentive[4] focuses on the importance of positive reviews on the app store, showing that "77% of percent of respondents reported that they read at least one review before downloading a free app".

Moreover, if reviews and stars on the app stores are related to users' judgment, and they are subjective, there are metrics automatically collected when using an application showing how the software performs in an unbiased way. As a matter of fact, Google included *Startup Time* among the metrics available on the *Android Vitals* platform, which is a tool whose goal is to monitor the most important characteristics for a released Android application, in order to identify possible issues and improve the current version. In particular, concerning the launch time of an app, the platform alerts the developers when detecting slow startup performance. The technical details of such tool will be discussed in section 3.5, but the key point is that achieving good results in the different aspects considered by the platform will result in raising the ranking of the application when performing a research on the Play Store, giving more visibility to the product and increasing the possibility of attracting new users.

[4] How to Improve App Ratings and Reviews

Chapter 2

Application startup

To understand the factors that influence the most the launch time of an application, it is necessary to first explain which kind of operations are performed when an Android application is started.

2.1 Creation of the application process

In the diagram reported in figure 2.1 we can observe the different steps that are performed by the Android system after the user taps on the application icon (launcher):

1. The system creates an `Intent`, which contains the information about the desired application;

2. The `Activity Manager Service` starts a new process that will deliver the `Intent` to the `Zygote` process. The `Zygote` process is a special process that "starts when the system boots and loads common framework code and resources"[4] (like framework classes and shared libraries);

3. The `Zygote` forks a new process whose main thread is the `Activity Thread`. The aforementioned `Zygote` initialization allows to reduce the total startup time without the need to instantiate objects that are common to the Android framework. The main thread has also an associated `Looper` which is in charge of processing messages posted on its `MessageQueue` and perform the corresponding actions;

   After forking the `Zygote` process, the system waits until the `Application` process is created, this event will be signaled via IPC (**I**nter **P**rocess **C**ommunication).

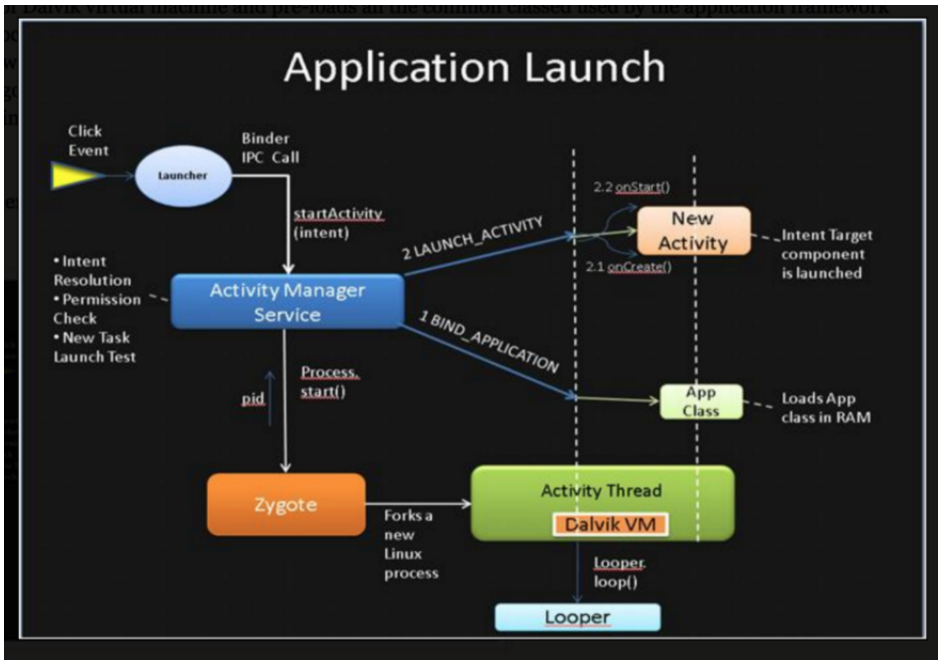


Figure 2.1. Stages of app launch from user click to activity launch. Source: Android Application Launch explained: from Zygote to your Activity.onCreate()

4. When the system is notified that the `Application` process is created, it makes an IPC call to `ActivityThread.bindApplication()` on the `ActivityThread`. This will cause the **main thread** to load the APK and instantiate the `Application` object through the `onCreate()` method;

5. After the application is created, the `Activity` corresponding to the `Intent` is created and, subsequently, started;

It is easy to notice that the events occurring until `bindApplication()` is called are automatically executed by the system and, and there is not much the developers can do to optimize this process. However, starting from the above mentioned method, app-specific information are processed and that's the point in which app startup monitoring should start.

We will observe this process more in details when analyzing the system trace of the **Mobile Banking** application startup (see section 5.2 for details).

2.1.1 The main thread

After having created the *Application* object, the system will launch the so-called main thread of the application, that will be responsible for handling most of the events happening when interacting with the app, like input events, or drawing the user interface. For example, when the user presses a button (corresponding to an input event), this action will be inserted into an event queue, whose events will be processed by the *Main* thread according to the order of insertion, meaning that a block of code cannot be executed until the previous one in the queue is completed.

2.1.2 Activity lifecyle

In the previous section have been explained the steps up to the creation of the `Activity`. In this paragraph, instead, will be presented the details of the `Activity` class lifecycle, that are useful to understand the different ways in which an Android application can be launched.

The `Activity` class offers a set of callbacks that allow the developer to observe changes in the status of the `Activity` itself, we will focus on three of them: `onCreate()` , `onStart()` and `onResume()`.

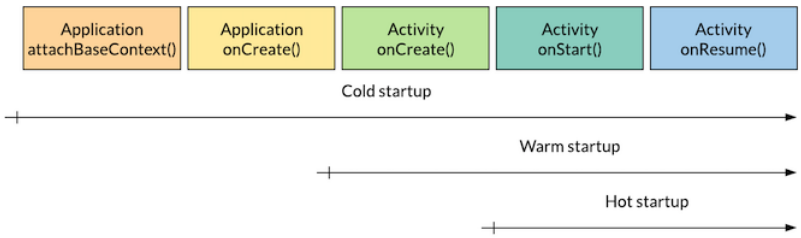


Figure 2.2. Operations performed in case of COLD, WARM and HOT start.

In the figure above, we can observe that after the creation of the `Application` object, the system sequentially executes the methods related to the `Activity` instance:

1. `onCreate()`: it is the first callback that gets executed when the system creates the `Activity` object. In this method, the developer typically embeds the initialization logic related to the specific activity, like loading information from previous activities, or inflating the `layout` associated

to it. More details about `layouts` in Android will be discussed in section 2.2;

2. `onStart()`: the activity enters the *STARTED* state, and the app prepares it to be brought to the foreground. At the end of this callback, the Activity enters in the *RESUMED* state;

3. `onResume()`: the Activity here becomes available to the user to interact with it.

2.1.3 Different states of application start

Although the diagram in figure 2.2 represents all the main operations that can be performed upon application launch, some of them may be skipped. In fact, an application can be started in three different ways:

- **Cold** start
- **Warm** start
- **Hot** start

Cold start

As we can observe in Figure 2.2, in the case of a **Cold start** the application must perform more operations compared to the **Hot** and **Warm start**. In such scenario the application object has not been created by the system, yet. This usually happens when the user launches the application for the first time after installation, or they start the app after rebooting the device, or simply after the application process has been killed by the system or the user (for example by clearing recent activities). In this case there are no traces of the application in memory, so everything must be performed from scratch.

Moreover, when the application is launched this way, the user is displayed the default blank screen until the corresponding activity is completely drawn, that can happen after several seconds. To mitigate the problem and offer the users a more "pleasant" waiting, many application introduce a custom **themed launch screen** showing, for example, the logo of the application itself.

It is easy to notice that this is the worst-case scenario, because it is the one that requires more time to be completed. For this reason, the thesis will focus on the analysis of the application start in the case of a *Cold* start, since

it ensures that the application process will undergo the same operations upon startup, and it also embeds the section of application startup that are likely to slow down the launch of the application process.

Warm start

In the *Warm start* case, the application can skip some operations that are, instead, performed on a **Cold start**. This happens when the application has been recently used, sosome information about it still reside in memory. From 2.2 we can see that the process needs to start again from the `Activity.onCreate()` method, avoiding, for example, the overhead caused by the creation of the `Application` object.

Hot start

Finally, **Hot start** is the fastest one, because the process can avoid creating bot the `Application` and the `Activity` object. This results in little overhead and a feeling of immediate launch of the application.

2.2 Layout in Android

In section 2.1.2 it was mentioned that, when the `Activity` enters in the CREATED state, the `onCreate()` callback gets executed, and usually it is the place where the corresponding `layout` is processed. In this section, we discuss the details of layouts in Android, as it will be necessary for understanding how a wrong implementation can lead to performance problems.

A `layout` defines the structure of the user interface in an Android application. It follows a hierarchy that has the main components as branches, and smaller ones as leaves. There are two types of elements in an Android layout:

- `View`: a view represents a single element in the layout the user can interact with, for example a text box or a button.

  Also notice that a View can still be present in the layout, but it is not visible to the user. In fact, the corresponding `visibility` attribute can assume three values[1]:

[1] Android Documentation - View

  - `VISIBLE`: the usual condition, the view is visible to the user;
  - `INVISIBLE`: the view is not visible to the user, but it occupy some space and it will be considered when positioning the other UI elements;
  - `GONE`: the view is invisible and it is not considered when calculating the position of other Views.

- `ViewGroup`: they are containers for multiple `Views` or other `ViewGroups`. For example, grids and lists are kinds of `ViewGroups`.

A sample layout tree is represented in the figure below:

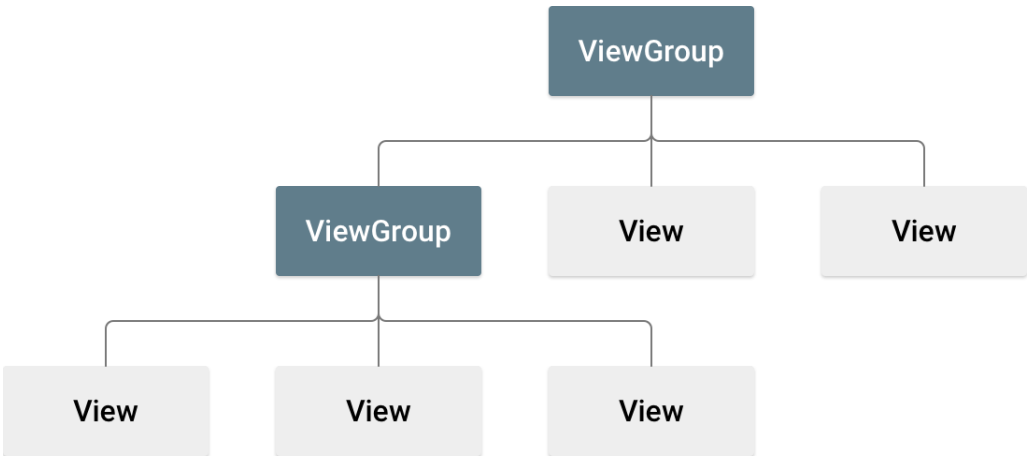


Figure 2.3. Example of layout hierarchy in Android.
Source: Android Developers - Layouts

Layouts can be defined either in a separate XML resource file or directly specified in the Java/Kotlin code when creating, for example, the activity that will host the layout. The first approach allows the developer to separate in a clearer way the application logic from the presentation, and it also makes it possible to reuse components across different screens/activities. However, it is also possible to declare layouts in separate files and then customize them at runtime.

2.2.1 Inflating a layout

It has already been presented that, usually, when the `onCreate()` callback of an `Activity` gets executed, it generally includes the task of processing the corresponding layout. This process is known as **layout inflation** and

it means to: consider the desired XML resource file, parsing it to create the `Views` and `ViewGroups` specified in the layout file and then adding the processed sub-tree to the layout hierarchy.

2.2.2 Common UI components in Android

LinearLayout

The **LinearLayout** is a `ViewGroup` that is used to distribute children components vertically or horizontally, by specifying the `android:orientation` attribute. In the figure below it is reported an a simple example of a vertical and horizontal `LinearLayout`:

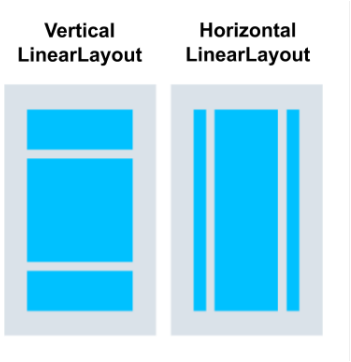


Figure 2.4. Example of usage of `LinearLayout` ViewGroup.
Source: Android Kotlin Fundamentals: LinearLayout using the Layout Editor

FrameLayout

This kind of `ViewGroup` is usually adopted to either display child `Views` in a stacked-way (for example a background image with a text area on top of it), or to contain a single child and positioning it in the UI.

RelativeLayout

The **RelativeLayout** is a `ViewGroup` that allows to specify the position of children `View` nodes with respect to each other (so, among siblings) or to the parent view (the `RelativeLayout` itself). However, the Google Developers team suggests to use `ConstraintLayout` for optimized performance instead of `RelativeLayout`.

ConstraintLayout

This `ViewGroup` can be employed to generate complex layouts, by leveraging the "constraints" of `View` objects relatively to siblings or the parent `ConstraintLayout`, similarly to the `RelativeLayout`. However, it is simple to use, as it is well integrated in the Android Studio ID and it also "provides similar functionality to `RelativeLayout`, but at a significantly lower cost".[6].

ListView

It is similar to the `LinearLayout`, but this `ViewGroup` is used when there are more elements the user may scroll (e.g. a home page in a social network) and only allows to arrange child elements vertically. Since the number of descendant components is determined ad runtime, this layout needs an `Adapter` that is used to specify the characteristics that children have, instructing the `ListView` on the number of the elements and how to correctly display them.

However, the current state of the art for dynamically-generated content is the `RecyclerView`[2].

RecyclerView

Similarly to the `ListView`, this layout is used to display scrollable list of elements whose appearance and number is determined at runtime. Unlike the `ListView`, though, it allows to distribute children components also horizontally, or in a grid-style layout.

Additionally, the `RecyclerView` offers performance improvement over the `ListView` as elements that are no more visible (for example due to the scrolling of the UI) are not destroyed, but rather re-used in order to minimize the overhead required to update the interface and populating it with new components.

Also in this case, the `Adapter` is in charge of specifying the number of elements and the content of the `RecyclerView`. Another important element in this kind of `ViewGroup` is the `ViewHolder`, which acts like an interface between the `Adapter` and the `View` that will be associated to an item in the list. There are three methods that need to be implemented when using a RecyclerView:

[2]ListView - Android Developer Reference

- `onCrateViewHolder:` here it is specified which layout is going to be associated to the children components (the `ViewHolder`);
- `onBindViewHolder:` after the desired layout for the children has been determined, it needs to be populated with data that are specific for each item in the list (e.g. the name of the contact in case it is an address book, or the corresponding avatar image).
- `getItemCount():` this method is needed to instruct the `RecyclerView` on the number of elements, since it can also be modified at runtime (for example, adding a contact in the address book).

WebView

This particular type of `View` allows to display a web page inside the interface of an Android application. After having inflated this component, the desired web page can be loaded using the `loadUrl()` method, that receives as a parameter a `String` corresponding to the address of the web resource.

Custom View

Although the Android framework offers several predefined components to build the user interface, like the ones previously presented in this section, there are cases in which the developer might need to define its own `View` to meet their needs (for example a custom animation, or a layout that would be difficult to achieve otherwise) these kind of components are called *Custom Views*.

To obtain the desired effects, it is up to the developer to define the way in which the Custom View is drawn, by specifying its behaviour through the callback methods made available by the `View` class, like `onMeasure()` and `onDraw()`, which will be discussed in the next section.

2.2.3 How Android draws the UI

Even though the layout shown in 2.3 is a simple one, it is easy to understand that the view hierarchy of the application can become quite deep, as components can embed other ones and additional elements can also be injected at runtime. This allows the programmer to create very complex and detailed interfaces ,however, the deeper a hierarchy is, the more likely it is to cause a performance overhead, due to the way the Android framework draws the interface.

In fact, for each sub-tree of the layout, the framework needs to perform three operations, starting from the top of the hierarchy:

1. **Measure**: the application traverses the tree in a top-down way. During this phase each `View` (or `ViewGroup`) determines the dimensions for itself (and for its children, in case of a `ViewGroup`) in a recursive way. Notice that a parent `View` may call `measure()` more than once on its children. This happens when there are many sub-components, or child `View`s do not specify the exact desired dimension, so a compromise between children's request and available space must be found;

2. **Layout**: also in this case the process performs a top-down traversal of the tree to position each child on the screen, using the information obtained during the previous stage;

3. **Draw**: after both the position and the dimension are established, the view is drawn on the screen.

2.3 Common problems in application startup

In this section will be presented the most common issues that can be observed when analyzing the startup performance of an Android application. It is convenient to classify them into two categories:

- layout (related to graphical aspect of the application)
- internal operations not related to visual components.

2.3.1 Blocking the Main thread

We have previously presented the role of the main thread in section 2.1.1, and it is simple to notice that, since the main thread is unique in the application context, it is important to avoid keeping it busy in performing long-lasting operations, because otherwise it won't be able to process other events and interactions coming from user activity. Moreover, as the access to UI components is not thread-safe (meaning that trying to modify an element of the user interface from other threads can lead to concurrency problems), performing modifications on the UI should only occur from the main thread. It follows that, if the main thread must process too much information, drawing events (and so visual updates on the screen) cannot be processed by the system,

causing the user interface to appear blocked and not responsive. If the UI remains "frozen" for more than 5 seconds, the system displays the well-known ANR (**A**pplication **N**ot **R**esponding) dialog, asking the user if they want to wait for the application to (try to) complete the operations causing this stall or immediately close it.

In the next sections, we will analyze which are the most common operations that can keep the main thread busy and that are critical for the application startup phase.

2.3.2 Heavy initialization of the Application

As we have discussed in section 2.1, after the application process is created, it becomes responsible for instantiating the `Application` object, and this is achieved by executing the `Application.onCreate()` method. Note that this step is needed only when the application is launched in a COLD start scenario, as this overhead is avoided in WARM and HOT start cases.

While in simple applications the system performs the application initialization automatically, and the process is transparent to the developer, in the case of more complex apps, the `Application` object may need to be customized to perform some operations that will influence the entire application flow. In fact, since the code embedded in the `onCreate()` method is going to be executed at the beginning of the application lifecycle, it is a suitable place to include initialization logic, not only for our application, but also for third-party libraries and SDKs (**S**oftware **D**evelopment **K**it is a collection of software tools that can be embedded into an application to provide a set of functionalities: for example **Firebase Performance Monitoring**, that will be presented in section 3.4, offers its SDK to allow collecting real time information from application usage).

However, if on the one hand it is convenient to move all the setup operations inside this block of code, on the other hand, every instruction executed at this point is going to cause a delay in the launch of subsequent `Activity`, increasing the time the user has to wait before being able to interact with the application.

The Android guidelines suggest that: complex `Application.onCreate()` method, disk or network I/O, deserialization (see section 2.3.4 for details) are among the most diffused causes of slow down at application launch time [3]. For this reason, the intuition is that, although some components need to be initialized right away when launching the application process, there may be others for which this kind of operation could be delayed to a second stage,

for example after the interface has been completely drawn. For these second category of objects, the solution resides in the so-called **lazy initialization**, meaning that one "should initialize only objects that are immediately needed".

2.3.3 Complex Activity initialization

Large layouts inflation

In section 2.1.2 it was presented the `Activity` lifecycle, highlighting that, when the application is launched in a COLD state, the first method that gets executed is `Activity.onCreate()`. Here, similarly to what happens in the case of `Application.onCreate()` the code executed includes initialization tasks for the activity itself.

However, differently from the `Application` object, the `Activity` is also related to the user interface, and the above-mentioned method is usually the place in which the corresponding `Layout` is inflated (presented in section 2.2.1).

Layout inflation is not, per se, a heavy task, however larger view hierarchies take more time to be processed, and this can lead to delays in the app launch time. A possible solution, coming from the Android Developers guidelines consists in: flattening the view hierarchy simplifying its structure and avoiding to immediately inflate parts of the layout that won't be visible, thus wasting resources.

However, since some of the delayed components may be needed at a later time (that could also happen shortly after the interface is drawn), the Andorid framework offers a particular component called `ViewStub`, that is specifically thought to act as a placeholder for UI elements that are not needed right away, but could become visible afterward. As reported in the Android documentation "`ViewStub` is an invisible, zero-sized View that can be used to lazily inflate layout resources at runtime"[8] via the `inflate()` method. After that, the `ViewStub` is removed from the view hierarchy and the corresponding sub-tree takes its place. The XML resource file where the stubbed view is described can be specified via the `android:layout` property.

Double taxation

When discussing the steps needed to draw the UI of an Android application in section 2.2.3, it was presented that the framework performs the `measure()` step to compute the dimension of the UI components, then the `layout()` pass

to establish the position of such elements, and finally the `draw()` method to actually draw the pixels that will populate the UI.

However, when the view hierarchy becomes deeper and more complex, it is possible that executing the `measure()` and `layout()` one time is not sufficient to acquire the information needed to also perform the `draw()` pass: in this case the multiple traversals of parts of the hierarchy are performed to establish the size and the position of the UI elements. This phenomenon is known as **double taxation** and is defined as "having to perform more than one layout-and-measure iteration"[7].

The overhead caused by the **double taxation** is not, per se, a serious problem, because it is normal for applications to have lots of components that contribute to deepen the hierarchy. However, when large parts of the visual hierarchy require more than one `measure()` and `layout()` pass, the performance penalty may be significant. For this reason, it**double taxation** represents a problem when the components causing it are placed near the root of the hierarchy, or there are many instances of the interested component, each one requiring more than one pass.

Double taxation can occur for several reasons, but the following are the most common cases:

- using a `RelativeLayout` will cause the framework to incur in **double taxation**, due to the intrinsic way this component is handled by the framework;

- using a `LinearLayout` with an horizontal orientation or that specifies *weights* (to establish the dimension of children in `LinearLayout` with respect to each other).

2.3.4 Heavy deserialization

It has been presented in section 2.3.2 that one of the common problems that arise when analyzing the startup phase of an application is the intensive usage of **deserialization**, which is the process of taking as input data structured in a specific format and converting it into an Object. In this section we will focus on the explanation of this operation, taking into consideration the JSON format, which is currently the most popular format for deserializing data.

The JSON format

JSON (JavaScript Object Notation) is a file and data interchange format that uses key-value pairs to store information in a human-readable way. Its grammar is very simple as the keys are always strings, while the values can have one of the following types:

- object;
- array;
- string;
- number;
- boolean (true/false);
- null.

An example of a JSON object could be:

```
{
    "name": "John", // string
    "surname": "Smith",
    "age": 45, // number
    "address": { // object
        "street": "Sample street",
        "postalCode": 12345,
        "city": "Sample town"
    },
    "phoneNumbers": [ // array
        123456789,
        987654321,
        135792468
    ],
    "married": false, // boolean
    "insuranceType": null // null
}
```

Figure 2.5. Example of a JSON object. Comments, which are not allowed in JSON format, are added in C-style for clarity.

Having briefly illustrated the structure of a JSON file, we can introduce the concepts of *serialization* and *deserialization*.

Serialization

It is the process of converting an object which resides in memory into a JSON string. In the context of the considered application, the object belongs to a *Class*.

For example, if the input for the serialization is the following object, belonging to the class *Person*:

```
public class Person {
 private String name; // "John"
 private String surname; // "Smith"
 private int age; // 45
}
```

the output will be:

```
{"name":"John","surname":"Smith","age":45}
```

Deserialization

It is the inverse operation with respect to *Serialization*, consisting in converting a JSON string into an Object. The deserialization in very much used when offering a customized experience to the user. In fact, data coming from network requests or locally saved into the memory of the device, are usually stored in a JSON/XML format, and since it is more convenient to deal with Objects rather than raw text, these information need to be deserialized into class instances.

If, on the one hand, this process is quite intuitive and there are several third-party libraries to support it, from the standpoint of performance it can end up being a time consuming task, influencing negatively the startup of an application.

Chapter 3

Tools

There are several tools that can help developers to analyze the performance of an Android application. The goal of this chapter is to provide an overview of the most common options and how they can be employed to collect relevant information about application performance and help to identify possible issues.

3.1 Android Profiler

It is a tool[1] that provides real-time data coming from the execution of an Android process. For apps that allow debugging, it offers the possibility to inspect four different aspects of the execution:

- CPU activity;
- Memory allocation and de-allocation;
- Battery consumption;
- Network activity (requests/responses).

Throughout the work, the CPU activity inspector has been the most used feature, because it allows the developer to have a quick overlook of the functions that are internally called by the application, also providing statistics about them. [2]

[1] link to Android Developers documentation

[2] More details about the Android Profiler here

3.1.1 CPU profiler

As the name suggests, the CPU profiler offers an overview of the CPU and threads' activity while interacting with the application. There are four settings that can be chosen to trace the operations performed:

- **Sample Java Methods:** it captures the call stack of the application at regular intervals (defaults to 1ms) to identify the methods that are called. The problem with this approach is that methods whose duration is very small could be skipped (i.e. if one method starts right after the captures and ends before the next one). However, it has been the preferred method for the analysis of the application, as it causes little overhead compared to other approaches.

- **Trace Java Methods:** in this mode, the profiler captures the beginning and the end of each method, this way, also short functions are recorded, so it is more accurate than the previous approach. However, tracing each entry and exit point of the various methods is very expensive, and the overhead seriously affects the performance of the application. An interesting comparison of how different profiling methods affect the runtime performance can be found at the following Can you trust time measurements in Profiler?.

- **Sample C/C++ functions:** keeps track of native functions called inside the application.

- **Trace system calls:** allows to visualize the state of execution of the different threads, showing also the activity per CPU core. Its most important feature is that it permits visualize custom sections that has been defined in code using the `Trace` class, and specifying the start of a section via the `beginSection()` method and the end of it via the `endSection()` method . Throughout this work, custom traces have been extensively used, as they allow the developer to highlight block of code that would not be shown using the previous methods.

  However, system tracing can also be used via command line, using the `systrace` or `perfetto` commands. In the following sections we will present their features.

To gain insights about the sequence of methods that are invoked during the application lifecycle, the Profiler offers the developer different graphs that are more or less suitable, depending on the focus of the analysis.

Call chart

Call chart: the default one, shows the methods invoked in chronological order on the horizontal axis. If one method invokes another one, the callee (the method that is invoked) is represented below the caller (the method that internally invokes the callee), as depicted in the following diagram:

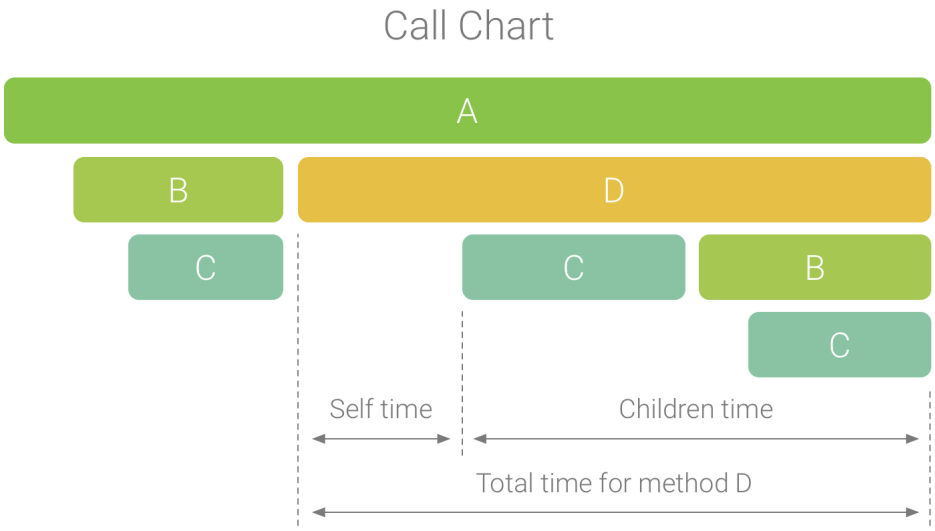


Figure 3.1. Diagram of a call chart in Android Profiler.
Source: Inspect traces using the Call chart

Throughout the application analysis, this has been the preferred method among the ones available in the Android Profiler, as it provides an easy way of tracing the order of the functions invoked and it also shows their duration.

Flame chart

A flame chart shows a graph that is inverted if compared with the Call chart in Fig. 3.1, in which identical method invocations are aggregated. For example, in Fig. 3.2 we can observe a call chart in which there are more methods that share the same call stack (B_1, B_2, B_3 and C_1, C_3).

In the corresponding flame chart, these methods will be aggregated to produce the following diagram: This kind of graph is useful to understand which methods are executed more times, and how they affect the total duration.

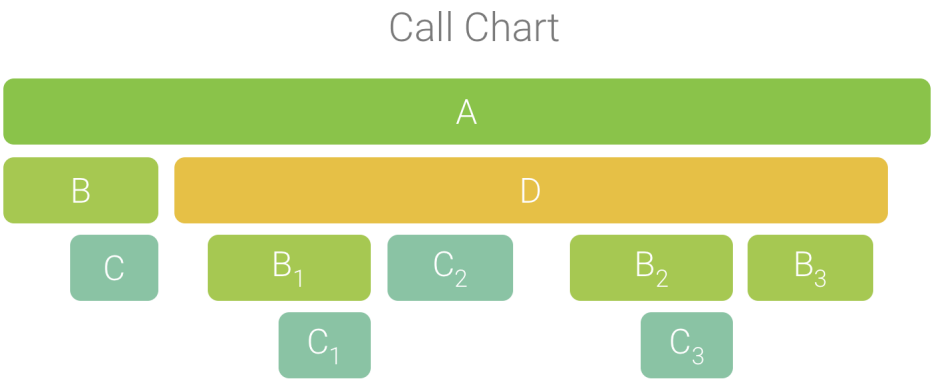


Figure 3.2. Diagram of a call chart with identical call stacks in Android Profiler.
Source: Inspect traces using the Flame Chart tab

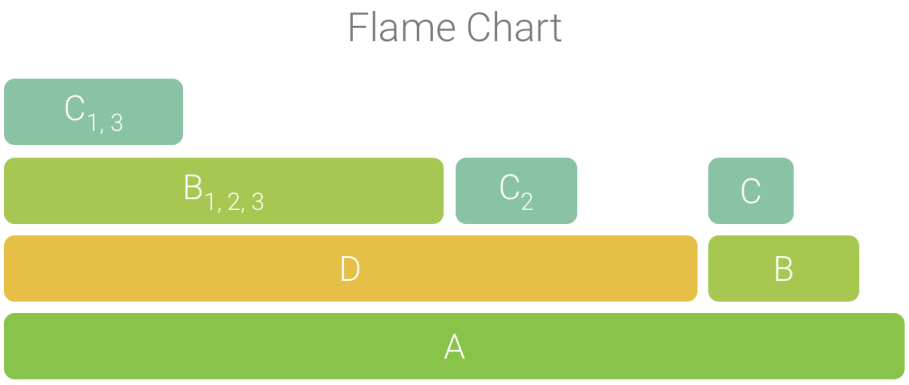


Figure 3.3. Diagram of a flame chart in Android Profiler.
Source: Inspect traces using the Flame Chart tab

Top down and Bottom up

They provide a list representation of the methods invoked during the selected interval. The **top down** representation displays and expanding list in which the root nodes are the callers and the leaves are the callees. The **bottom up** representation, instead, starts from the callers as root to end up with the callees as final nodes of the list. These graphs are useful because they offer the developer the possibility to directly access the snippet of code corresponding to the invoked method (if this is inside the project's source files).

3.1.2 Memory profiler

It is a component of the *Android Profiler* that permits to trace memory allocations and de-allocations, as well as heap status and garbage collection events.

Heap and heap dump

The **heap** is a memory area from which the resources needed for object instantations are allocated. Capturing a **heap dump** means to record a snapshot of all the objects that reside in memory at a certain point in time.

Heap dumps are particularly useful when looking for possible memory leaks, as they allow the developer to identify objects that are still in memory, but that are likely unused. Since the dimension of the heap is limited and decided by the system, if there is no free space to allocate new objects (or arrays) in memory, the memory management environment will free resources via *garbage collection*.

Garbage collector

Recall that the *garbage collector* is a process managed by the memory management environment (so it is generally transparent to the programmer) whose goal is to identify resources (objects) that can no longer be used in the future by an application and release those resources to make room for other object instantiations.

Although garbage collection is a reasonably fast operation, if the application makes an improper use of memory by instantiating too many objects, this can lead to performance issues. In fact, the developer does not directly control when *garbage collection* occurs, so it is the system that decides when to perform it, and to do so, user process must be temporarily paused. Depending on the amount of memory required by the application, *garbage collection* can take more time to be executed, and it can lead to noticeable delays in the execution.

3.1.3 Energy Profiler

The Energy Profiler is another component of the Android Profiler that monitors the usage of different resources (like CPU, location sensors, network radio), indicating how each of these activities affects battery consumption.

3.1.4 Network profiler

Concerning network activity, this tool permits to inspect the network requests performed when executing an application. This is not only important for debugging reasons, like detecting errors or the thread where these operations are performed, but it also allows the developer to identify possible problems to the frequency of such activities. In fact, as briefly introduced in the previous section, frequent networks requests may lead to a greater battery consumption, as turning on the network radio hardware or keeping it constantly active consumes extra battery power.

3.2 System Tracing

System tracing means to "record device activity over a period of time" [5]. Differently from CPU profiler, explored in section 3.1.1 it traces the activity of all the processes running on the device, not only the selected application, including information about CPU cores' status (frequency, running processes, scheduling).

As stated in the previous section, the Android Profiler allows to record a system trace from the GUI, however, throughout the work, this kind of analysis has been performed using the `perfetto` and `systrace` command line tools.

3.2.1 Systrace

Systrace is a python script provided by the Android SDK tools that allows the developer to capture a system trace, and, when the recording is complete, it generates a report in HTML format, that can be inspected via the Systrace viewer or Perfetto UI.

3.2.2 Perfetto

Similarly to `systrace`, it records the device activity, but it offers the possibility to collect longer traces, while keeping a reduced dimension of the generated reports. It is the preferred way of collecting application activity information employed in this work, since it also opens directly the recorded trace file inside the Perfetto UI.

3.2.3 System tracing app

Another alternative to capture a system trace is to use the *System Tracing App* directly from devices running Android 9 (OS API level 29) or higher. The advantage of this approach is that there is no need to connect the mobile device to the PC and it also allows to share the trace file after the recording is completed. However, traces must be opened in a Systrace Viewer to be inspected, so the `perfetto` approach has been preferred as it performs this step automatically.

3.2.4 Perfetto UI

It is a tool that allows the developer to record, open and analyze system traces directly in the browser. It supports both the formats recorded using `perfetto` and `systrace`.

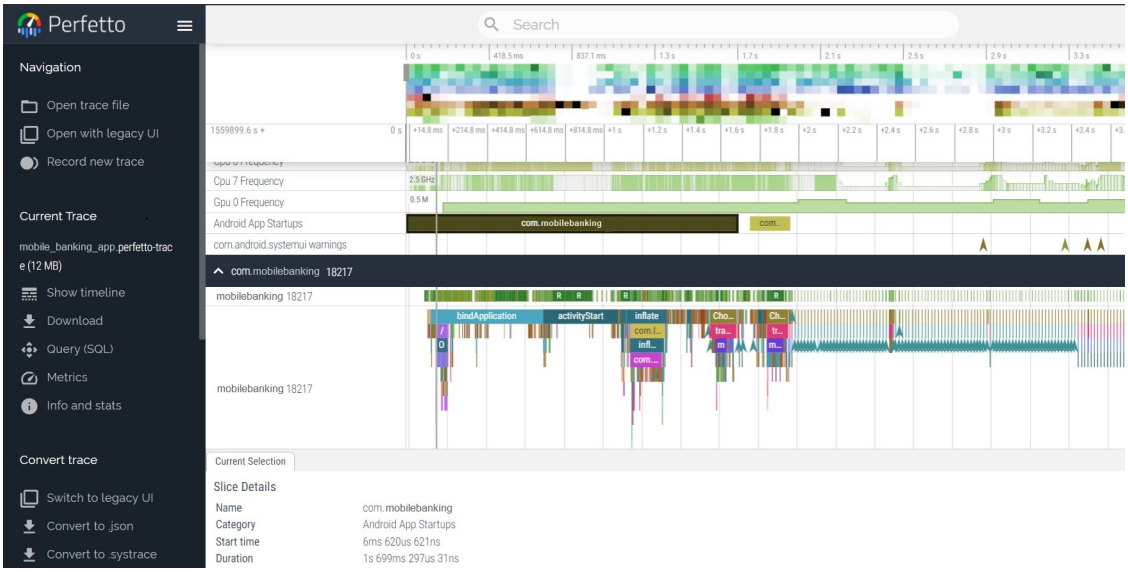


Figure 3.4. System trace showing the Mobile Banking app startup in Perfetto UI.

In figure 3.4 it is possible to observe how a sample trace recorded with `perfetto` or `systrace` looks like when opened in the Perfetto UI. On the left there is the main menu of the tool, that offers several features, like:

- Open a trace file: even though the command-line tool automatically opens the trace in this interface, this is convenient to open several traces in multiple tabs of the browser to compare the behaviour of the application in different cases;

- Download the currently opened trace: this feature is handy because the script invoked via the command line could overwrite previously recorder traces, so saving a copy of it may be useful to avoid losing it.

On the right part of the interface, the timeline of the trace is displayed, and the user can navigate the trace by zooming into a particular section, or select a different portion of the trace. Each section of the trace is highlighted in a different color, making it easier to locate it.

On the top part of the interface are displayed the information about CPU state, in particular it is possible to observe which process is executed on each CPU core and the corresponding frequency. Moreover, considering the startup of an application, it is possible to notice that there is a separate row (named *Android App Startups*) where it is reported the section corresponding to the the startup of an Android application (if any) and the corresponding name of the package it belongs to (in this case *com.mobilebanking*) . The section follows the definition of **Application Startup**, starting from the creation of the *Intent* and ending when the first frame of the application is drawn.

Below the above-mentioned rows, that are related to system-wide information, there is a list of expandable tiles, each of them corresponding to a particular process/application. In the example reported in figure 3.4, we can see that the application selected is part of the package *com.mobilebanking* and the corresponding process has been assigned the pid (process id) 18217.

After having expanded the tile of a particular process, on the left are displayed the name of the thread and the corresponding *tid* (thread id): in case of the *Main thread* the thread id is the same of the process id and the name is the same as the package the application belongs to (in this case *com.mobilebanking*.

As we can observe from the figure, for the selected thread there are two rows:

- the top one captures the status of the thread, that can assume four different values:
  - `RUNNING`: the thread is performing some work and actively running on CPU;
  - `RUNNABLE`: the thread is ready to run but it hasn't been scheduled yet;
  - `SLEEPING`: the thread needs some resources that are not currently available. When they will be released, the scheduler will put the

thread in a `RUNNABLE` or `RUNNING` state;

- `UNINTERRUPTIBLE SLEEP:` it is generally associated to I/O operations, the application hasn't received the result back, yet.

This row is useful to identify possible scheduling issues.

- the bottom one, instead, captures the call stack of the corresponding thread. Each section corresponds to the invocation of the `beginSection()` method, where nested calls will produce deeper stacks.

Another handy feature is the search bar at the topmost part of the interface, that is useful when the developer needs to search for a specific Trace (e.g. a custom one). If one or more matches are found, the tool highlights the corresponding sections.

Furthermore, the interface allows to select a larger portion of the recorded trace, which possibly includes more sections (or call stacks), and displaying aggregated metrics for the corresponding slices. This is particularly useful in case there is a method (traced via the `beginSection()` call that is repeated multiple times during the execution of the application, or multiple slices that have been given the same name. This way, the developer can gain interesting insights like: the number of occurrences, the total aggregated duration, and the average duration of each call.

Finally, when a section is selected by clicking on it, the tool shows the details of the current slice, where we can find the name of the slice, the start time and its duration (along with other less relevant information).

3.3 Logcat

The traces recorded with `perfetto`, `systrace` and the Android profiler have been utilized when there was the need to inspect more in detail a precise section of execution, because their analysis permits to understand how long a particular section requires to be completed and the call stack corresponding to it. However, as we mentioned in previous sections, these tools lead to an unavoidable performance penalty, that depends on the method chosen for the analysis, which is heavier in the case of the Android Studio profiler. For this reason, to observe how the startup time was affected after having introduced some modifications, the *Logcat* tool was used.

In this window, directly available on Android Studio, the software displays messages coming from the application, managed by the `Log` class. The `Log` class offers several methods to send messages to the *Logcat* windows, like:

- `Log.d` to display a message for debugging purposes. This method is generally used to signal the beginning or completion of a method, to signal that a certain point in code has been reached;

- `Log.e` to display an error message;

and this helps the developer to trace the execution of the application and identify possible errors without the need to start it using the *debugger*.

Concerning the application startup analysis, when executing the application on devices running Android 4.4 or higher, the *Logcat* window automatically outputs the `Displayed` metric. This value represents the time elapsed between the launch of the process and the completion of drawing the first frame of the corresponding activity. In the case of the **Mobile Banking** application, the interested activity is the *HomePageActivity*, and this is the output displayed in the logcat window when launching the application:

```
Displayed com.mobilebanking/.HomePageActivity:+1s371ms
```

This information can also be obtained by starting the application using the `adb shell Activity Manager` command:

```
adb shell am start -W -n
```

`com.mobilebanking/.HomePageActivity` The output produced after executing such command is the following:

```
Status: ok
LaunchState: COLD
Activity: com.mobilebanking/.HomePageActivity
TotalTime: 1371
WaitTime: 1376
Complete
```

As it is possible to notice, the output contains the *state* in which the application was launched (in this case *COLD*), the activity that was launched and the total time it took to complete the operation.

The possibility to start an application from the command line has been essential for the analysis of the startup performance, as it allowed to produce a script to repeatedly launch the application and collect the metrics of interest, as presented in the next section.

3.3.1 Script for multiple runs

Since the Logcat `Displayed` metrics report the launch time for one execution, to tackle the variability in the results obtained, a script for performing repeated measurements was produced, using Windows Batch Scripting. It is reported below:

```
for /l %%x in (1, 1, %1) do (
 adb shell am force-stop com.mobilebanking
 sleep %2
 adb shell am start -W -n
 com.mobilebanking/.HomePageActivity
 | grep "TotalTime"
 | cut -d ' ' -f 2
)
```

The script works as follows:

1. the process of the corresponding application is killed (if it is running). This is done to ensure that the application is launched in a *COLD* state;
2. the system awaits for a number of second that is user-specified;
3. the application process is launched;
4. the metric corresponding to the `Displayed` time is reported, filtering out redundant information using a regex[3] to extract the value of the field.

A greater number of iterations allows to reduce the variability in the results obtained, while the waiting time between one run and the other has been introduced to avoid that some elements could be reused and influence the total launch time.

3.4 Firebase Performance Monitoring

It is a tool that collects statistics about application performance in real-time directly on end-users's devices, then those metrics can be visualized and analyzed in the Firebase Console. Among the aspects that are monitored by default when the SDK is integrated into the application, there are:

[3] a regex (**Reg**ular **Ex**pression) is a sequence of characters that specify a pattern. More information can be found here

- App startup time;
- Frame rendering time:
- HTTP requests.

Additionally, the software allows to define custom traces (similarly to what has been presented in section 3.1.1) to capture specific sections of the execution. Also in this case, we can establish the beginning and the end of a trace leveraging the `Trace` class and the corresponding `start()` and `end()` methods. Moreover, the SDK offers the possibility to monitor the execution of an entire method using the `@AddTrace` annotation, that can be more convenient that manually defining the beginning and the completion of a trace. Furthermore, for each trace, it is possible to define several metrics to be collected (besides the **duration**, that is the one enabled by default) within the trace scope.

Differently from the methods presented in the previous sections, **Firebase Performance Monitoring** is essential when there is the need to collect large-scale data that reflect user experience when interacting with the application. For example, the details of a trace can be analyzed considering different aggregation level, automatically managed by the SDK, like:

- **App version:** to compare the behavior of the current version with respect to previous ones;
- **Country:** useful to verify that the location does not affects performance;
- **OS API level:** whether older versions of the Android OS may lead to performance issues;
- **Device:** visualize the difference when running the application on high-end low-end devices, as well as smartphones coming from different manufacturer;
- **Radio:** how the metrics are affected when using WiFi rather than mobile data.

Regarding this work, the **Firebase Performance Monitoring** tool has been used on a sample application to understand how the platform works and how it can be used on a real case. However, the access to the platform for the considered application was reserved to developers with higher privileges. Nonetheless, the data presented in tables 1.1 and 1.2 have been extracted from this tool.

3.5 Android Vitals

Similarly to Firebase Performance Monitoring (presented in section 3.4), **Android Vitals** is a tool developed by Google to collect metrics related to app performance from the activity of end-users "who have opted in to automatically share usage and diagnostics data"[9] when using the application. Then the data collected are aggregated to allow the develop team to later analyze them via the Google Play Console.

The kinds of data that can be collected via the **Android Vitals** tool are the following:

- Battery usage;
- Stability;
- App startup time;
- Rendering time;
- Permissions.

Regarding the **app startup time** metrics, **Android Vitals** collects startup times for the application when it is launched in a **cold**, **warm** or **hot** state (see section 2.1.3 for further details).

Among the details that can be analyzed in the Google Play Console there is the **impacted session** metric, that indicates the "percentage of sessions during which users experienced a slow start-up time for each respective system state"[9]. A slow start-up time is detected when the application launches in more than:

- **5 seconds** in the case of a **cold start**;
- **2 seconds** in the case of a **warm start**;
- **1 second** in the case of a **hot start**.

These values can be considered as the upper limit for a reasonable app startup time.

3.6 Android Studio Layout Inspector

As discussed in section 2.2, the layout hierarchy of an application can easily become more and more complex, especially when the application offers many functionalities to the customer, each one requiring a component the user can interact with.

When the structure of the visual tree is defined in a single resource file it is easy to understand how the different components interact between themselves and their relation inside the hierarchy. However, as more and more `Views` or `ViewGroups` are added to the tree (and/or are instantiated at runtime rather that only using XML files), keeping track of the overall structure of the user interface can become complicated.

To simplify the work, the **Layout Inspector** tool, integrated in Android Studio, offers the possibility to visualize in real time the layout tree of a running (and debuggable) Android application.

In figure 3.5 it is reported the interface of the tool when attaching layout inspection to the **Mobile Banking** process.

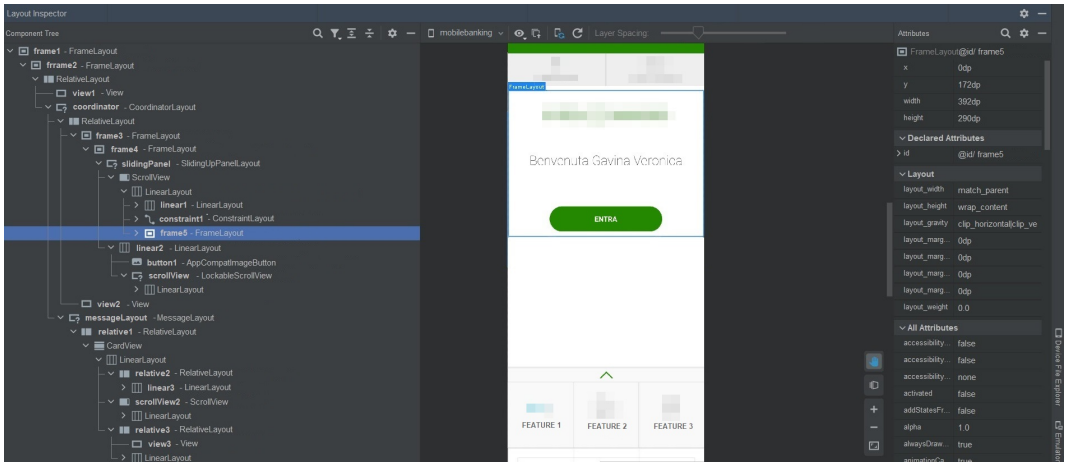


Figure 3.5. The interface of Android Studio Layout Inspector.

As it is possible to notice from the screenshot, on the left pane of the interface the developer can find the **Component Tree**, that reports the hierarchy of the components in the current screen, which is represented in the middle section of the tool. A convenient feature is that, by clicking on a component in the tree, the corresponding view in the interface is highlighted, making it easy to locate a certain `View` or `ViewGroup` (in the example, the element named `frame5` has been selected). Moreover, on the right side of

the UI are reported the attributes of a selected component, like its dimensions, the identifier for the element, the corresponding resource file and its accessibility characteristics (e.g. visibility).

This last feature is very useful because it can help the analyst in identifying layout elements that are not displayed in the interface, but that still require computational power to be processed, possibly wasting precious time upon startup. Additionally, the tool permits to display the layout tree in 3D, and it can be handy to better understand the components' stack, as well as their position in the screen.

The Layout Inspector has been intensively used when performing the analysis of the application at a visual level (reported in chapter 5).

Chapter 4

Mobile Banking application

As already introduced in section 1.1, the application considered for this case study is the **Mobile Banking** Android app. Like other mobile banking applications, it offers the customer the possibility to perform several operations that are accessible through the different screens of the application. The goal of this chapter is to briefly present what is the execution flow of the app at launch time, to let the reader better grasp the context of the application launch.

When the user taps on the launch icon, they will trigger the set of operations illustrated in section 2.1. In this case, the Activity that is going to be started is named `HomePageActivity`, and after the first frame of the application is draw, the user is displayed the interface reported in figure 4.1:

From the screenshot, we can notice that the user experience is already customized, in fact the interface includes a "welcome view" with the full name of the user that has performed the access. This is due to the fact that, throughout this work,we will be considering the case in which the user has already logged into the application, and has decided to preserve the information (by enabling the famous "REMEMBER ME" option) so that future accesses will be faster, because this is the most common scenario.

The interface can be divided into three sections:

- at the top there are two buttons that allow the user to receive support when using the app;
- in the central part there is the "welcome view" with the "ENTRA" button to perform again the login (for security reasons) and access all the

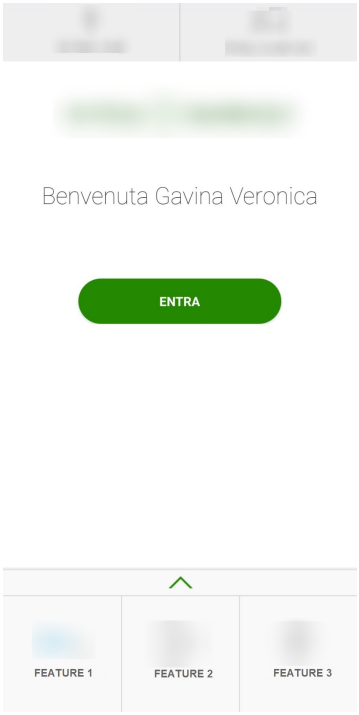


Figure 4.1. Home page of the application in the most common use case, when the user agrees on remembering the login information for future accesses.

functionalities of the application;

- at the bottom there is a sliding panel from which the customer can quickly access the features that are most commonly used.

Chapter 5

Layout Analysis

The first part of the analysis conducted on the **Mobile Banking** application is focused on performance problems that are related to visual aspects, like layout resource files and the corresponding views. Sections 5.1 and 5.2 will introduce, respectively, the structure of the user interface with the corresponding resource files and an overview of the most important sections captured through a system trace.

After having introduced these background concepts, a deeper analysis of the major layout resources will be conducted in section 5.3, with the goal of identifying possible issues that may slow down the application startup. Finally, a plausible solution to optimize these critical parts will be discussed in section 5.4.

5.1 Application structure

Before being able to analyze the launch performance of the app, it is necessary to understand the structure of the application and present the resources that are involved.

Considering the *HomePageActivity* screen, reported in 4.1 and using the Layout Inspector presented in section 2.2, it has been possible to identify the layout structure at launch time, which is reported in figure 5.1:

Starting from the Android Hierarchy Root, the layout resources files that are involved upon application startup are the ones highlighted in blue, whose (partial) sub-tree is reported in the boxes below them. Here the role of these components is summed up:

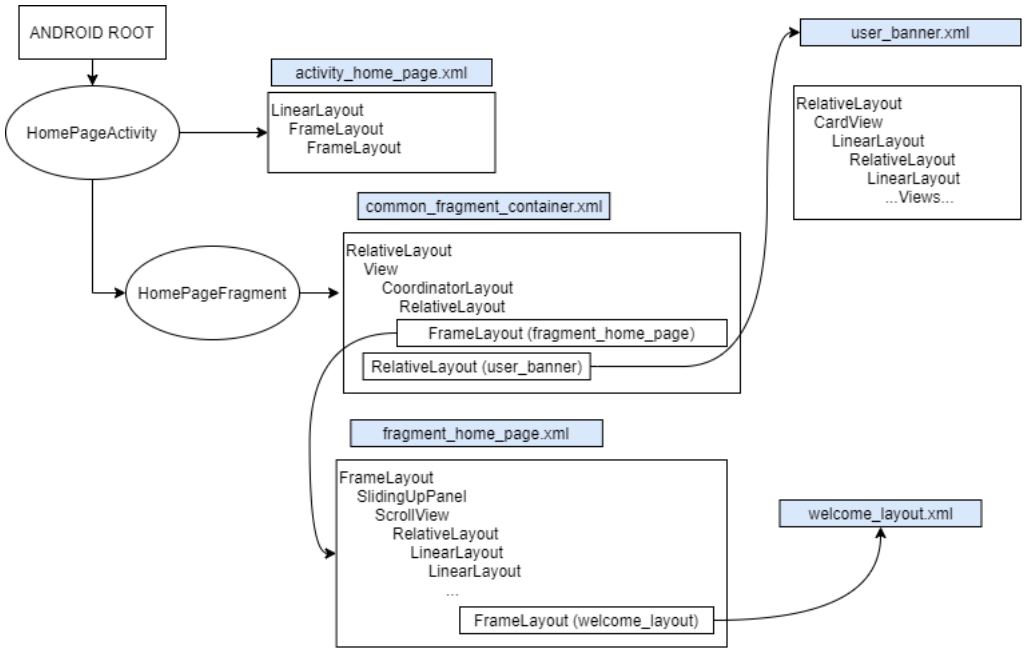


Figure 5.1. Graph reporting the structure of the layout after the first frame of the application is rendered.

- **`activity_home_page.xml:`** the main container for the *HomePageActivity*. This is the top-level sub-tree when considering application launch;
- **`common_fragment_container.xml:`** as the name suggests, this component is reused among different screens of the application (not only the initial one) and has the role of coordinating children components of the interface;
- **`fragment_home_page.xml:`** it includes UI elements related to the landing page, like the expandable panel and the welcome view;
- **`user_banner.xml:`** this element of the launch UI is used to show information and notifications to the user.
- **`welcome_layout.xm:l`** it embeds the "welcome view" the user is displayed when launching the application. It may also contain financial details if the customer has enabled the corresponding option.

Although not all the components are listed in the figure, the main modifications to the layout have been performed on these files.

5.2 First analysis of a system trace

The next step to gain useful insights on high-level actions happening at launch time is to use the `perfetto` script to collect a System Trace. Here it is reported the recorded trace:

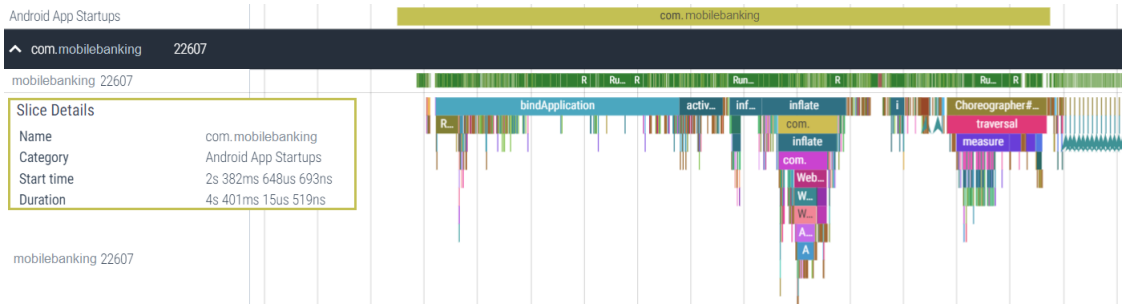


Figure 5.2. `perfetto` trace captured at application launch, and displayed in the Perfetto UI.

We can notice that the the tool reports the *Android App Startup* section, and by selecting it, we can observe that its total duration is 4.4s.

Using the **Slice Analyzer** panel of the Perfetto UI it is possible to visualize a in detail the timing information of the most relevant sections:

Name	Wall duration (ms)	Avg Wall duration (ms)	Occurrences
	7740.03453		2063
bindApplication	2086.839375	2086.839375	1
inflate	859.085104	53.692819	16
traversal	701.969792	350.984896	2
measure	607.066406	202.355468	3
RV OnBindView	320.239169	40.029896	8
com. mobilebankiing.AdvertisementWebView	236.916719	236.916719	1
activityStart	148.780781	148.780781	1

Figure 5.3. Slice details for the application startup in Perfetto UI.

Since the sections are sorted in descending order of *Wall duration* (the aggregate duration of slices with the same name) we can notice the following:

- `bindApplication:` this is surely the most consistent part of the application startup (about 2 seconds), requiring almost half of the total time needed for launching the app. As discussed in section 2, the operations occurring here are related to the creation of the `Application` process, and they are executed before the `Activity` is crated. Being this section not related to visual components, but rather to the initialization of

the `Application` object, the analysis of this part of the startup will be conducted in the next chapter;

- `inflate:` as presented in section 2.2.1, this method (which is repeated 16 times for the selected slice) is in charge of binding the `Activity` to its layout, either loaded from a XML file or dynamically created in Java/Kotlin code. It is evident that spending 860 ms for this kind of task can be a symptom of a large overhead during startup, and that the main thread could spend time in inflating many components, that may not be immediately required by the interface altogether;

- `traversal:` presented in section 2.2.3, the traversal operation represents the top-down analysis of the view hierarchy that the Android framework performs when it needs to draw the user interface. This operation encompasses the three stages of `measure()`, `layout()` and `draw()`.

- `measure:` as explained in the previous point, the `measure()` operation is the first step that the framework executes to draw the interface. Its consistent duration (607 ms) may suggest that the view hierarchy is too deep, and the high number of components causes a significant overhead when performing this first pass, due to the **double taxation** phenomenon (see section 2.3.3 for details).

- `RV onBindView:` the RV stands for `RecyclerView`, the `ViewGroup` presented in section 2.2.2. Recall that the `onBindViewHolder` is executed after the `onCreateViewHolder` to populate the layout of each element inside the view with the specific data;

- `AdvertisementWebView:` this particular `View` takes a huge amount of time to be inflated, it will be important to analyze whether it is immediately needed or it could be inflated at a later stage;

Moreover, by adding a custom trace at the start and at the end of the `onCreate()` method of the `Application` class, we can take a closer look to what happens before it is executed. The corresponding trace section is reported in figure 5.4.

In section 2.1 have been illustrated the steps that need to be performed when launching an application from scratch. In figure 5.4 we can observe that, after the beginning of the *App Startup Time* section, the system creates the main `ActivityThread` and other operations are executed before entering the `Application.onCreate()` method. In the reported trace it is

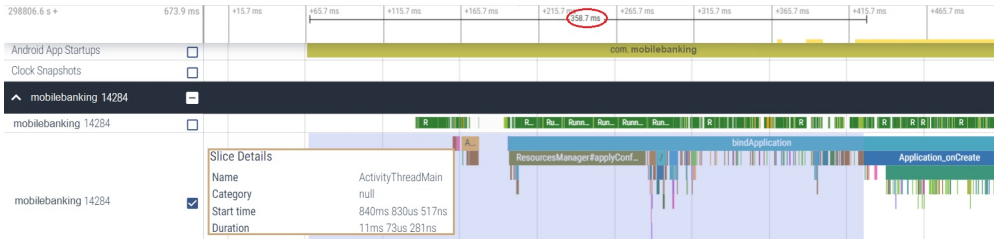


Figure 5.4. Slice details capturing the execution before `onCreate()`.

also possible to notice that approximately 358 ms elapse before beginning the customization of the `Application` object, meaning that a considerable part of the startup time is spent for operations that are not under the control of the developer, but they are rather executed by the Android framework, so it is important to observe that this overhead cannot be avoided when launching an application in a COLD state.

5.3 Identify problems

5.3.1 Analysis of `welcome_layout.xml`

Savings layout

Using the Layout Inspector tool, it has been possible to identify some elements in the launch screen layout that are not (always) visible at runtime:

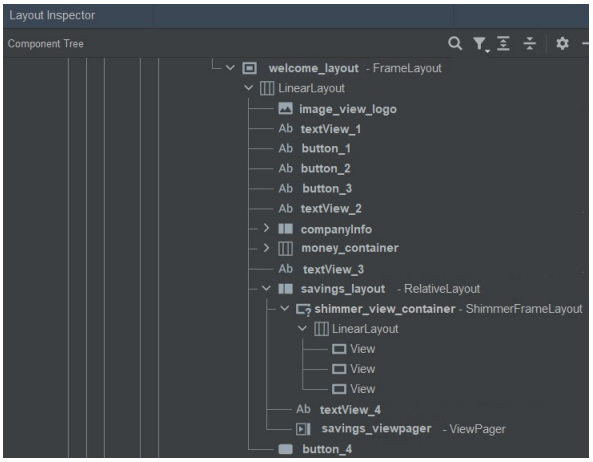


Figure 5.5. Welcome view layout in the landing page as seen in Layout Inspector.

In figure 5.5 we can observe the view hierarchy corresponding to the "Welcome view" that is displayed when the user reaches the Home page of the app. Among the components listed in the Layout Inspector, only three of them (the logo, the text with the customer of the user and the login button) are visible to the user, suggesting that the other ones may be unnecessary.

That's the case of the `savings_layout`, which is displayed only if the user has a particular option enabled. To understand the impact of inflating these components we can use the `perfetto` trace viewer.

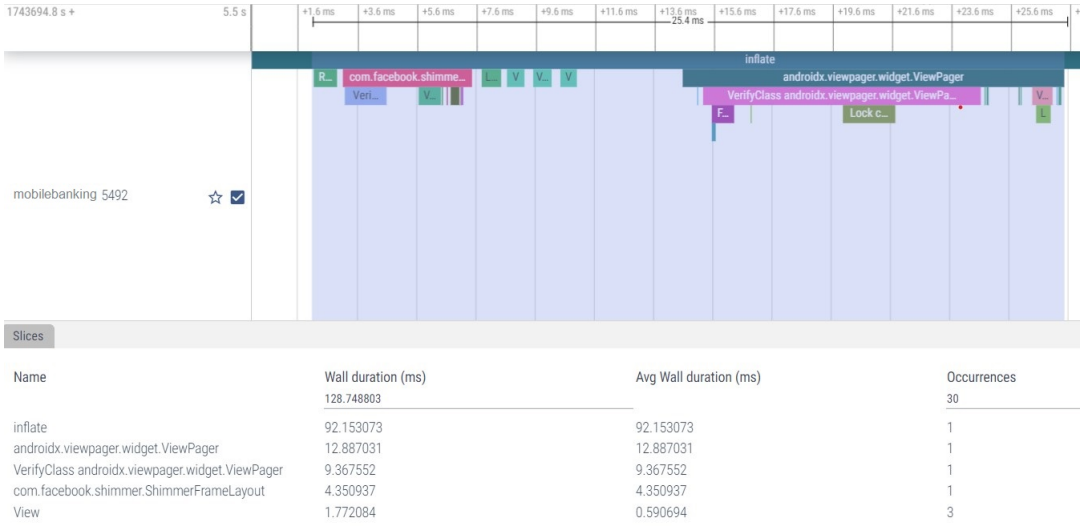


Figure 5.6. Detail of trace showing the impact of loading the money-box layout component.

In figure 5.6 it is possible to select the time interval corresponding to the inflation of such elements, that are listed in the bottom part of the interface. We can notice that to load these components in the UI (although they are not visible) it takes about 25 ms. A possible strategy to limit the impact of these component at application launch will be discussed in section 5.4.1.

5.3.2 Analysis of `fragment_home_page.xml`

In this section we are going to analyze the layout issues related to the `fragment_home_page.xml` resource file, which is one of the major layouts involved in the application startup.

By looking at the trace reported in figure 5.2, the first thing to consider is clearly the `inflate` section. To better understand the timing of the inflation process, we can make use of custom traces (presented in section 3.1.1) to

identify the components that are involved in this operation and how long it takes to insert them into the general view hierarchy.

AdvertisementWebView

The first component to take into consideration is the `AdvertisementWebView`, which, as highlighted in section 5.2, takes a considerable amount of time to be inflated.

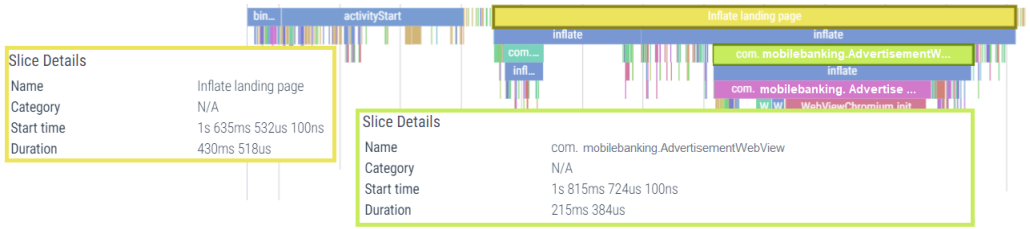


Figure 5.7. `perfetto` Startup section showing the impact of the WebView.

Using the Android Layout Inspector tool, we can recognize that this view is declared inside the hierarchy of the `fragment_home_page.xml` resource file, and that it is an example of *Custom Views*. The corresponding sub-tree is reported in the screenshot below:

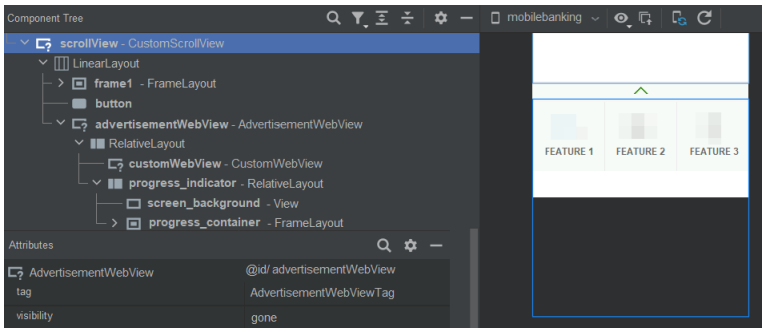


Figure 5.8. `AdvertisementWebView` hierarchy as reported in the Layout Inspector tool.

From figure 5.8 we can notice that the component extends the `RelativeLayout ViewGroup` and it includes two children:

- a `CustomWebView`, that, in turn, extends the `WebView` layout (presented in section 2.2.2) and will be used to display the web page;

- a progress indicator, whose details will be discussed in the next section.

Moreover, on the right side of the interface we can observe that the `AdvertisementWebView` should be placed inside the sliding panel with the different features: however, there is no trace of such component even when expanding the panel, in fact, its visibility is set to *GONE*, as we can see from the "Attributes" panel, reported below the component tree for clarity.

Although the effort to inflate this view is considerable, by analyzing the source code it has been possible to understand that the `WebView` is displayed only under certain circumstances (in particular if there is a commercial message for the user), otherwise it is not visible. A possible solution to mitigate this problem will be presented in section 5.4.2.

Home page progress

In the same layout sub-tree (of the `fragment_home_page resource file`) a component named `progress_indicator` is inflated at launch time. This element is defined in its own XML resource file, as it is shared across different sections of the application, and, as the name suggests, it is used to let the customer know that the application is performing some operation, so they have to wait for its completion.

However, using the Layout Inspector tool to analyze its view hierarchy (reported in figure 5.9) it is possible to notice two details:

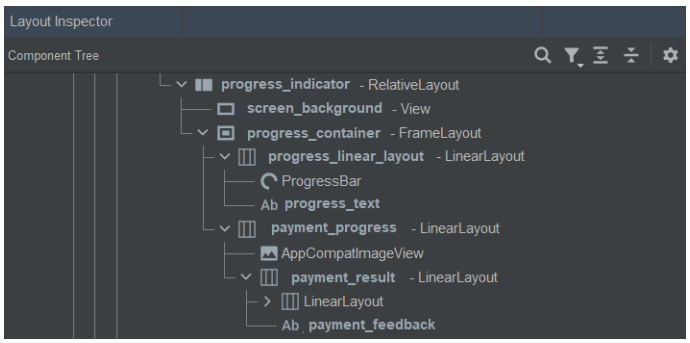


Figure 5.9. Progress indicator view hierarchy as displayed in Layout Inspector.

- even though the progress indicator seems a simple component, it has been customized, as it embeds several elements to achieve the desired effect, and the corresponding view hierarchy is quite nested.

In fact, the types of `View` and `ViewGroup` are highlighted on the right part of each element, and we can observe that there are 6 levels of nesting (because the `LinearLayout` further includes 5 images) having the `RelativeLayout` as root element. Moreover, The `LinearLayout` embedding the images needed for the animation of the loading indicator has a horizontal orientation. In chapter 2.3.3 we have seen that both these characteristics are among the causes of the **double taxation** phenomenon, which adds overhead when the interface must be rendered.

- the progress indicator view is not immediately displayed, and its visibility is set to `GONE`. Moreover, analyzing the code, it has been found out that when the `HomePageActivity` executes the `onResume()` lifecycle callback (see section 2.1.2 for details), the visibility of these component is set again to `GONE`.

To understand the effort needed to load this component, we can analyze a system trace detail, that captures the total inflation time for all the components belonging to the `progress_indicator.xml` resource file, which is reported below:

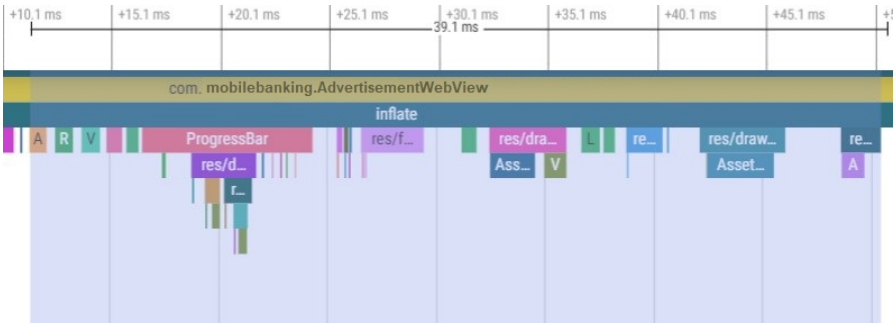


Figure 5.10. Inflation details for the `progress_indicator` component.

It is possible to notice that this layout sub-tree is first inflated when loading the `AdvertisementWebView`, presented in section 5.3.2 (as previously said, it is a common component that is re-used several times across the different screens of the app), and it is requested again shortly after.

Moreover, the impact it has on the startup time is non-negligible (about 39 ms), so a corrective action must be taken. The chosen approach to reduce this overhead will be discussed in section 5.4.3.

Sliding panel

In section 2.2.2 it was introduced the `RecyclerView` layout, which is used to display a list of elements that can be arranged in different ways on the screen. In the screenshot below, we can see how this component is used to build part of the interface of the **Mobile Banking** application at launch time:

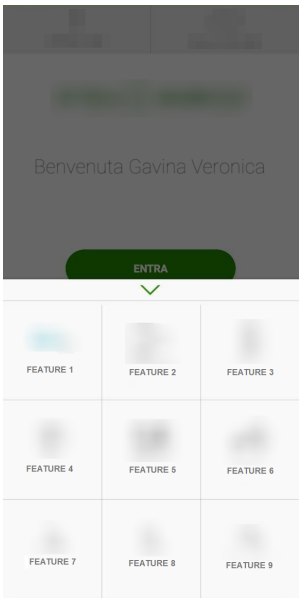


Figure 5.11. Expanded panel with multiple elements inside a `RecyclerView`.

When the user slides up the panel (that initially displays just three elements), this UI component is expanded, showing also other items that are initially hidden. The trace section below shows the time needed to display the `RecyclerView` embedded in the sliding panel:

Inside the blue rectangle are reported the statistics for the inflation phase of all the elements in the grid. It is also possible to notice that `inflate()` is invoked by the `onCreateView()` callback: recall that `onCreateView()` is used to specify the layout that each item in the `RecyclerView` is associated to. Since there are just nine items in the grid and they are all similar among themselves (an icon and a description), the 72 ms required to process the multiple layouts may suggest that additional work is performed.

Using the Layout Inspector tool, we can observe the structure of each element inside the grid, which is reported below:

In the screenshot reported in figure 5.13, we can observe that each element

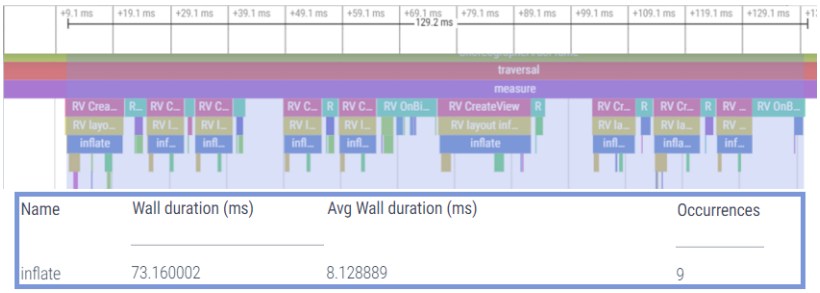


Figure 5.12. Section showing the time to inflate the RecyclerView.

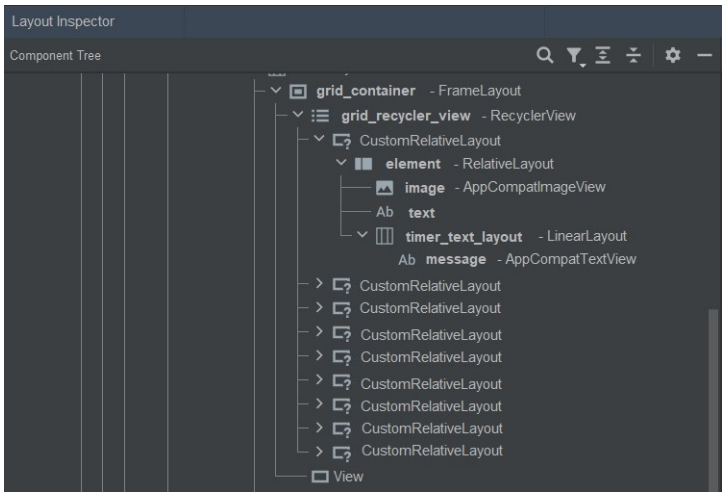


Figure 5.13. Structure of the elements of the `RecyclerView` as shown in Layout Inspector.

of the grid has a layout named `CustomRelativeLayout` as root element. This is an example of *Custom View*, presented in section 2.2.2, and it is used to obtain a `RelativeLayout` whose height and width are equal. The immediate descendant is another `RelativeLayout` that contains the visual elements of each grid item, in particular: an `ImageView` that corresponds to the icon of the functionality, a `TextView` containing the name of the service and a `LinearLayout` whose purpose is to contain a "timer" `TextView`. By further inspecting the source code, it has been possible to understand that the timer is needed for the *FEATURE 3* functionality, with the goal of alerting the user in case the time left for the booked service is running out. However, the timer element is needed only in one case out of nine, and this adds useless overhead to the inflation. A possible optimization of this layout component

will be introduced in section 5.4.5.

Previously in this section, it has been presented the role of the sliding panel that is initially collapsed. This component is not part of the standard Android library, but its documentation can be found at this link. The instructions presented in the *Usage* section suggest that this `ViewGroup` should only have two direct children: the first one will act as the main view, visible when the panel is collapsed, while the second one will contain the layout sub-tree corresponding to the sliding panel.

We can leverage the Layout Inspector tool to identify which components are involved for this purpose:

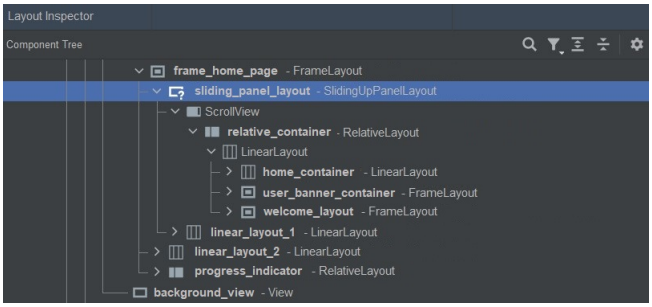


Figure 5.14. View hierarchy of the Sliding panel.

Inspecting the `CoordinatorLayout` class (that is a `CustomView` belonging to the `common_fragment_container` resource file), a reference to the `SlidingUnPanelLayout` was found: when the callback `onFinishInflate` is executed, the above-mentioned component is invoked to set the draggable portion of the layout via the `setDragView` method. By adding a custom trace embedding the invocation of this method, we can see the impact it has on the startup time:

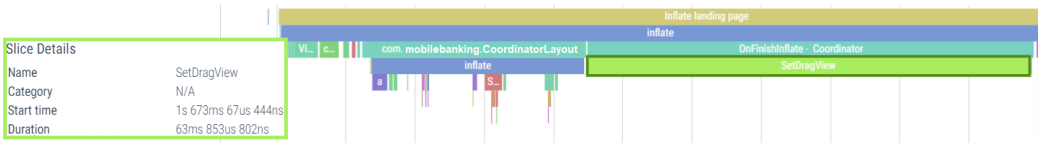


Figure 5.15. Detail of trace showing the section corresponding to `setDragView`.

The total time to dynamically modify the component is considerable (about 64 ms): a possible countermeasure will be discussed in section 5.4.5.

5.3.3 Analysis of `common_fragment_container.xml`

User banner

Taking into consideration the `common_fragment_container.xml` resource file and using the Layout Inspector, we can identify another component that appears to be hidden at launch time but it is still inflated. The element is named `user_banner` and its purpose is to show a banner where users can give a feedback about the application. In the screenshot below it is reported its layout tree:

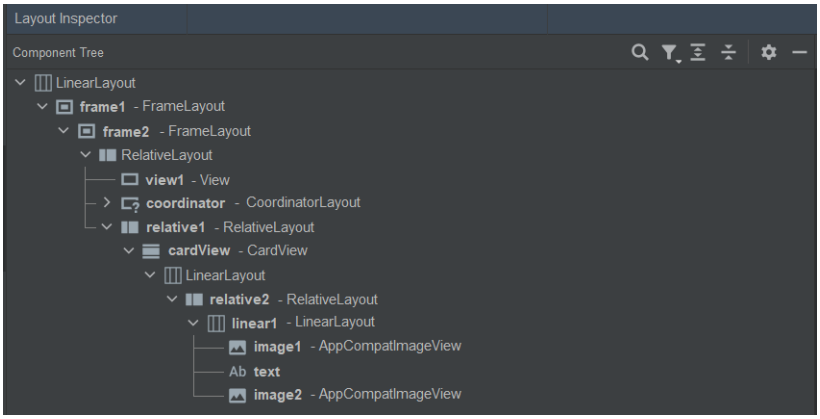


Figure 5.16. View hierarchy of the banner for user banner.

There is no trace of such element in the UI displayed at application launch, so we can try to identify the inflation of these components in the `perfetto UI` to understand the impact upon startup.

The corresponding trace section is reported below:

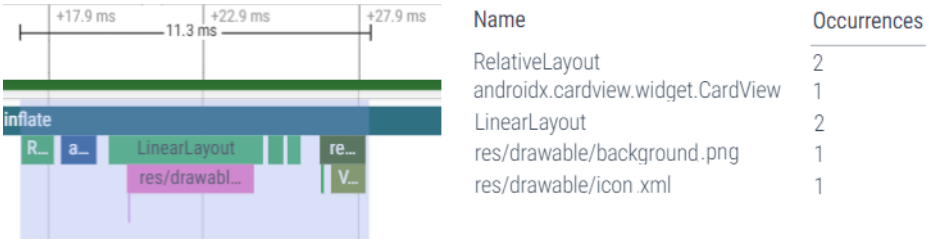


Figure 5.17. Trace detail for the `user_banner` component.

We can notice that the time required to load this component is approximately 11 ms, and, even if it is small compared to the total application

startup time (4.4s), it is important to keep in mind that every part of the view hierarchy adds complexity to the final layout, and this can increase the overhead caused by the **double taxation**. In fact, from image 5.16 and observing the source code of its XML file we can recognize that this component embeds a `RelativeLayout` and a `LinearLayout` with an horizontal orientation, that are both factors that contribute to the **double taxation** phenomenon.

To mitigate this problem, a plausible solution will be proposed in section 5.4.4.

Promotional message

Further analyzing the `common_fragment_container.xml` resource file, we can find that another custom component named `PromotionalLayout` is defined. Accessing the source file of this class, it has been possible to understand that this custom view extends the `RelativeLayout`. Using the Layout Inspector tool, we can also identify its view hiearchy, which is reported below:

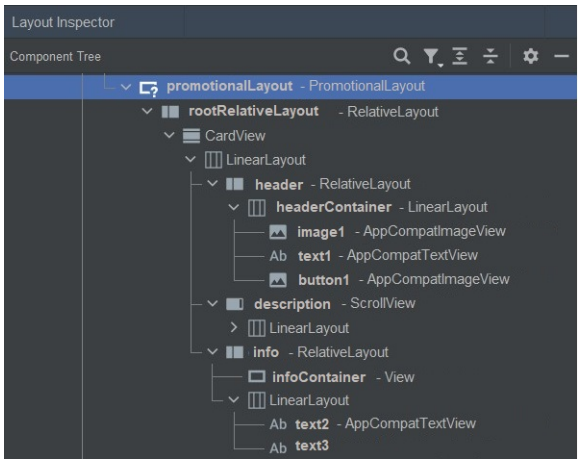


Figure 5.18. Elements of the `PromotionalLayout` view hierarchy.

The first thing that one can notice is that the `rootRelativeLayout` is its only direct child, so it is duplicated. Moreover, the hierarchy beneath this root element is sufficently nested, embedding other `RelativeLayout` (which we know not to be the best choice in terms of performance) and several `LinearLayout` presenting also an horizontal orientation. To better inspect the effort needed to inflate this component at launch time, we can define

the "*Inflate promotional message*" custom trace, captured in the following screenshot:

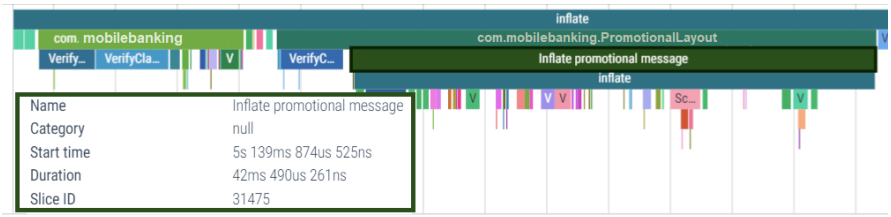


Figure 5.19. Trace detail for the `PromotionalLayout` component.

Is is possible to observe that it takes approximately 42 ms to load this element inside the view hierarchy at launch time, furthermore, it contributes to make the view tree deeper, increasing the overhead of the **double taxation**. Since it is not immediately visible when the user starts the application, a countermeasure must be taken to reduce its impact on the startup time: it will be discussed in section 5.4.6.

5.4 Proposed solutions

This section will be dedicated to the discussion of possible solutions for the problems that are related to the application layout hierarchy, identified in section 5.3.

5.4.1 Savings layout

The `Savings` layout is only needed if the user has enabled a certain option, otherwise it is inflated for no reason. The most intuitive solution consists in substituting the sub-tree corresponding to this component with a `ViewStub` (presented in section 2.3.3), whose inflation adds practically no overhead. At runtime, we can check whether the layout must be loaded or not, and if yes, call the `inflate()` method to replace the `ViewStub` with the original `SavingsLayout`. This approach allows to delay the inflation of this component only if needed, possibly saving about 25 ms (see section 5.3.1 for further details about the `SavingsLayout`).

5.4.2 AdvertisementWebView

It has been underlined in section 5.3.2 the huge impact this component has on the total launch time of the application (approximately 215 ms). Since it is not always immediately needed, but its visibility depends on the advertisements that the user may (or may not) receive, the natural solution here is to adopt the same approach presented in the `SavingsLayout` case (see previous section for details) and replace this heavy component with a `ViewStub`.

The new corresponding trace is reported in figure 5.20:

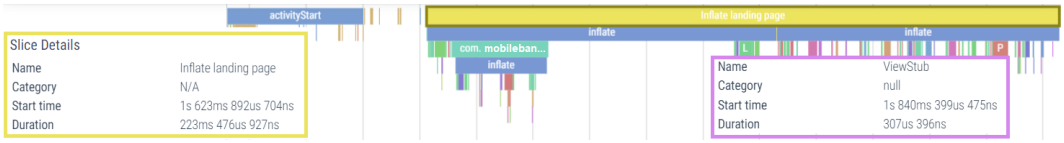


Figure 5.20. `perfetto` trace after removing the `WebView`

We can notice that the inflation of the `ViewStub` inflation adds practically no overhead to the total time needed for processing the layout (only 307 us).

5.4.3 Home page progress

We have seen in section 5.4.3 that not only this components is injected multiple times upon startup (one for the `AdvertisementWebView` and later for the `HomePage`), but the time needed to inflate its sub-tree is considerable (about 39 ms). Moreover, the fact that it has several levels of nesting contributes to add complexity to the general view hierarchy.

The proposed solution to this problem is to use, also in this case, a `ViewStub` to delay the inflation of this sub-tree only when needed. In the following snapshot it is reported a slice containing a simulated inflation on demand of the `progress_indicator` layout.

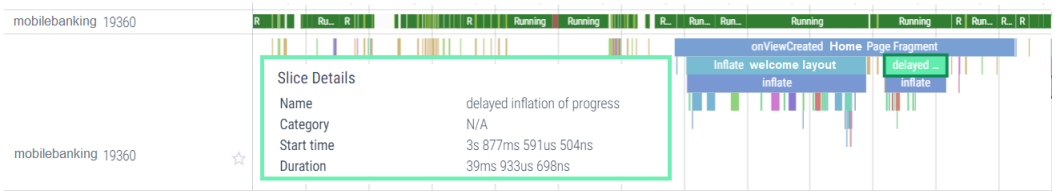


Figure 5.21. Detail of delayed inflation of the landing page progress.

It is possible to notice that the inflating the corresponding layout on demand requires approximately the same amount of time as immediately inflating the sub-tree.

5.4.4 User banner

Having identified another component that may not be immediately needed upon startup, we can consider, also in this case, to replace the corresponding sub-tree with a `ViewStub` to avoid useless inflation and complexity in the general view hierarchy, the estimated gain (when the component is not needed) would be of approximately 15 ms.

5.4.5 Sliding panel

In section 5.3.2 have been presented two issues that are related to the `SlidingUpPanel` component:

- The inflation of the same layout for all the nine elements of the sliding panel adds useless overhead, since it includes children that may be needed only in one case out of nine, increasing not only to load them into the layout sub-tree but also the complexity of the general view hierarchy;
- dynamically setting the `dragView` attribute seems to cause a substantial overhead (approximately 65ms).

Regarding the rendering of each feature layout, there are two things that can be:

- the root `CustomRelativeLayout` has only one child component, another `RelativeLayout`. It is possible to remove this second layer;
- the `LinearLayout` and the `TextView` needed to render the timer are used only in one case out of nine, meaning that their inflation is useless in the other eight cases. Recalling what presented in section 2.2.2, we can modify the `onBindViewHolder()` method by exploiting the `getItemViewType()` to check whether the element that is currently processed needs a certain kind of layout rather than another one. The idea is to define a "basic layout" resource file, common to all the elements inside the `RecyclerView` and another one that is specifically thought for the **FEATURE 3** element, that also includes the *timer view*. This allows to remove the (repeated) inflation of useless elements in the UI.

This is the simplest solution, an alternative could be to replace the subtree with a `ViewStub` and load it only if needed. This second approach permits to avoid loading the "timer layout" if not needed (while in the previously discussed solution it is inflated in any case).

In the trace below it is possible to see the inflation time for the modified version of the `RecyclerView`, adopting the first approach to mitigate the "timer layout" problem:

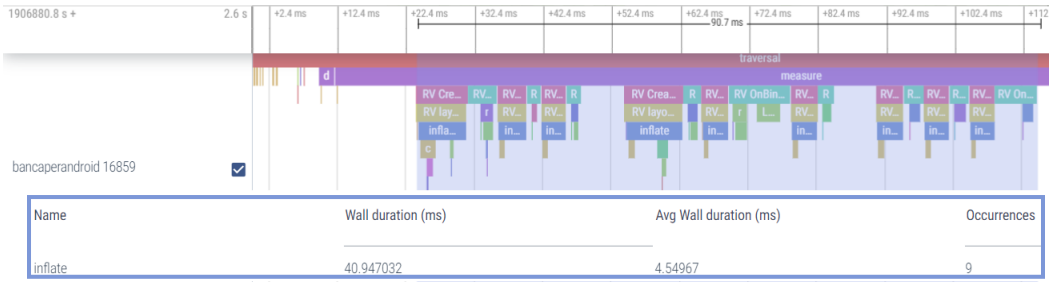


Figure 5.22. RecyclerView inflation time after modifications.

Comparing it with figure 5.12 it is possible to notice that the total inflation time is noticeably reduced (from 73 to 40 ms).

Considering now the overhead introduced by setting the `dragView` of the `SlidingUpPanel` at runtime, by looking at the documentation of the component, we can see that we can either indicate the draggable section at runtime or directly in the XML resource file using the `umanoDragView` attribute and directly specifying the id of the element we want to make draggable. Since the `setDragView` is invoked only in this case and there are no other references in the code, we can follow this approach.

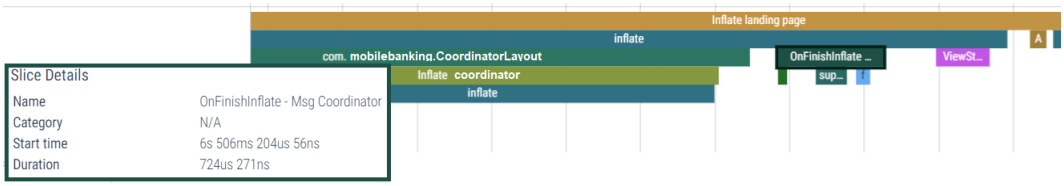


Figure 5.23. Detail of system trace showing the section corresponding to `onFinishInflate` after setting the drag view in XML.

In the figure above it is reported a second trace, showing the time needed to inflate the component using a static XML attribute. We can see that

the time needed to execute the `onFinishInflate()` callback is substantially reduced (from more than 63 ms to 724 us).

5.4.6 Promotional message

Having presented in section 5.3.3 the fact that this `CustomView` may not be needed upon startup and the corresponding tome to inflate it is consistent (about 40 ms) the most intuitive solution is to use a `ViewStub` to delay its inflation to a later moment in time (if ever needed).

Chapter 6

App internal operations

In this chapter will be presented the operations that slow down the application startup but that are not directly related to the UI.

6.1 Identify problems

6.1.1 Decryption

It is normal that mobile banking applications (like many others) must guarantee a high level of security to the customer to protect its sensitive information. Besides a secure communication channel with the application server for data that must be sent across the internet, one common approach to protect personal data that are stored on the user devices is to use encryption and decryption algorithms.

In the context of the **Mobile Banking** application, there is a special class which is in charge of handling security-related operations, we will call it `Security`.

By using the Android Studio CPU profiler to track the application startup, it has been possible to discover that the `decrypt` method is invoked 26 times upon application startup. Then a custom trace was added to estimate the time needed for the total invocations of such method via the `perfetto` tool. The corresponding trace is reported in figure 6.1.

We can see that a considerable part of the application launch time is spent in performing decryption. This operation cannot be avoided for security reasons, but by debugging the execution, it was discovered that some of the decryption algorithms were applied for data that were already decrypted shortly before, thus obtaining the same output.

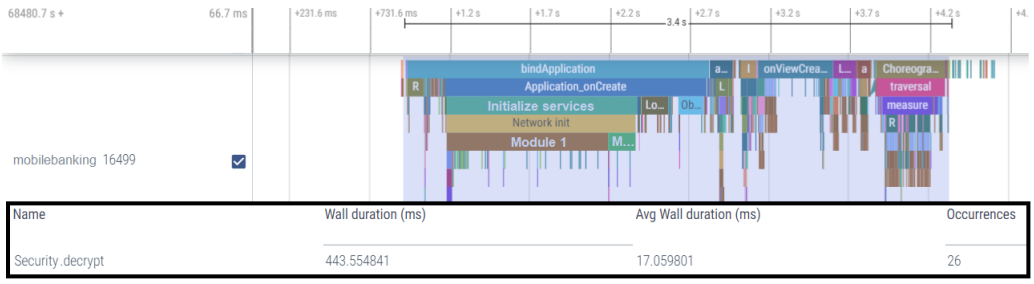


Figure 6.1. Trace showing the high number of invocations of the decryption function.

A possible solution to reduce the overhead coming from the repetition of the heavy decryption operation will be discussed in section 6.2.1.

6.1.2 Deserialization

In section 2.3.4 have been illustrated the key concepts of the JSON data format, as well as its usage to convert from a human-readable source into Java instances. Moreover, in section 2.3.2 it has been introduced that intense deserialization operations at application startup may increase launch time. In this section will be presented the major operations that are related to JSON deserialization, highlighting how they affect the application launch and what kind of solutions may be adopted to mitigate the problem.

Activity creation - Savings

When conducting the Layout Analysis of the application, it was illustrated in section 5.3.1 that there was a component named `savings_layout` which was not displayed at launch time, but it still took some time to be inflated into the view hierarchy. This element is used if the customer has enabled the option to visualize the balance of its "piggybank" even before having performed the access (it still requires to have logged into the application and having enabled the "remember me" option).

Using the Android Studio CPU profiler it was highlighted that the lifecycle callback `onViewCreated()` of the `HomePageFragment` class was requiring a considerable amount of time to be completed, due to a heavy deserialization operation. To obtain more accurate information about the above-mentioned method, a custom `Trace` was defined to capture its execturion and analyze it in the `perfetto UI`:

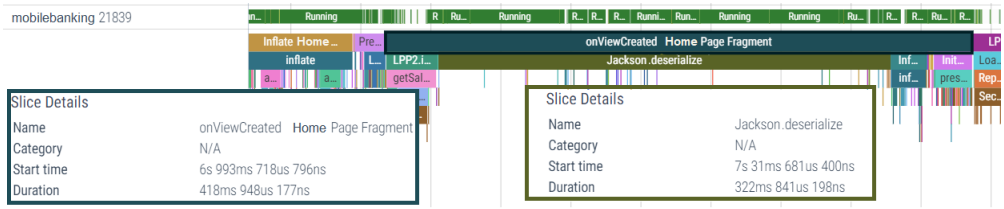


Figure 6.2. Trace showing the impact of deserialization inside the `onViewCreated()` lifecycle callback.

From figure 6.2 it is possible to notice the whole `onViewCreated()` method takes about 420 ms and most of the time is spent in JSON deserialization. Moreover, as previously said, it may also be that the user did not enable the option to show their "piggybank" balance right after launching the application. For this reason, it is important to understand whether this operation is necessary and if it could be delayed at a second stage or improved. A presentation of a possible approach will be discussed in section 6.2.2.

Cache Handler

It is common for applications to store data coming from a response to a request to a remote server and *cache* them to avoid issuing new identical requests in the future. Considering the case of the **Mobile Banking** application there is a special module that is delegated to handling this kind of data, that are needed upon application startup, and it is named `CacheHandler`.

By using the Android Studio CPU profiler it has been possible to track the execution flow of the operations carried out by this class, that are (partially) reported in figure 6.3.

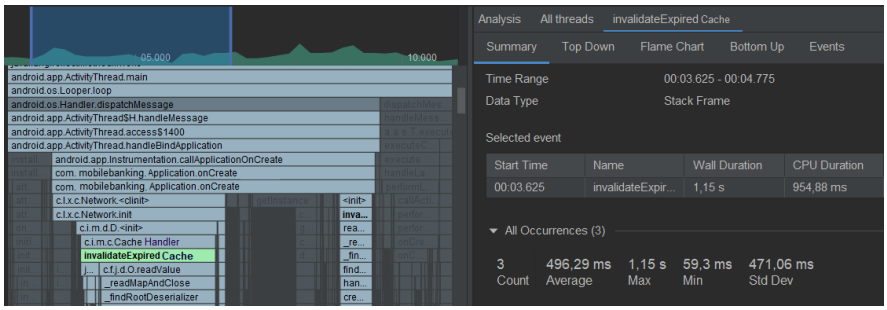


Figure 6.3. Call chart of the `CacheHandler` class initialization.

We can notice that the highlighted section is invoked inside the `init` method of the `CacheHandler` class. From the bottom section of the call stack it is clear that there is a deserialization ongoing. Moreover, on the right part of the interface it is possible to observe that the selected event is executed three times upon application startup, and the total duration of each call is surely not negligible (1.15 s as maximum and 59.3 ms as minimum).

As the name `invalidateExpiredCache` (which will be referenced to as `invalidate` for simplicity) suggests, when the `CacheHandler` object is instantiated (three times during the application startup) this method is executed to remove from the cache the responses that have "expired" according to a certain rule (whose details are not discussed).

By inspecting the source code, we can understand the execution flow for the above-mentioned method, which is reported below:

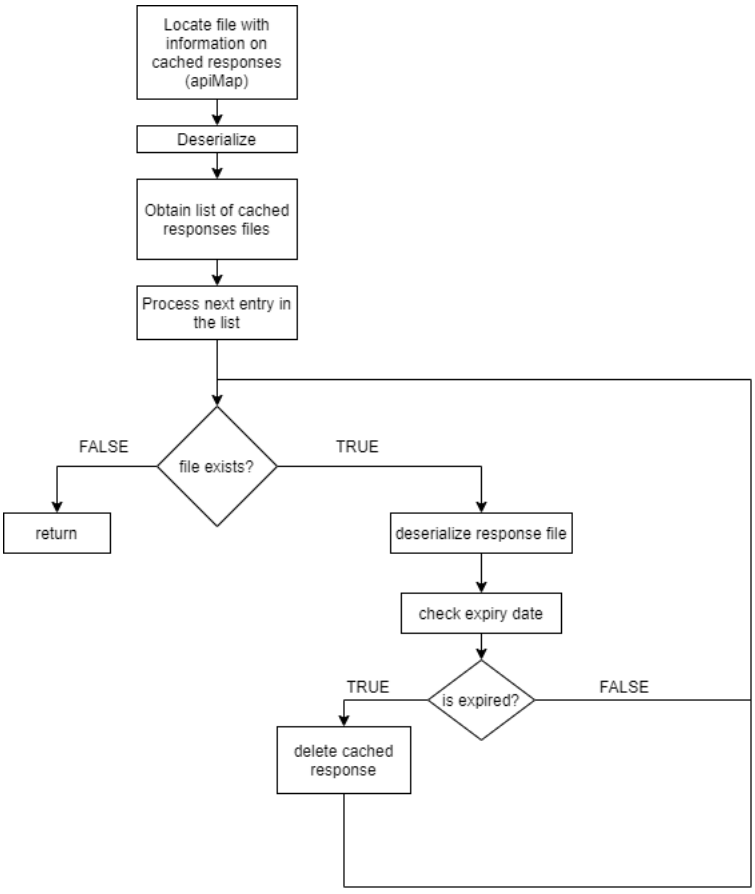


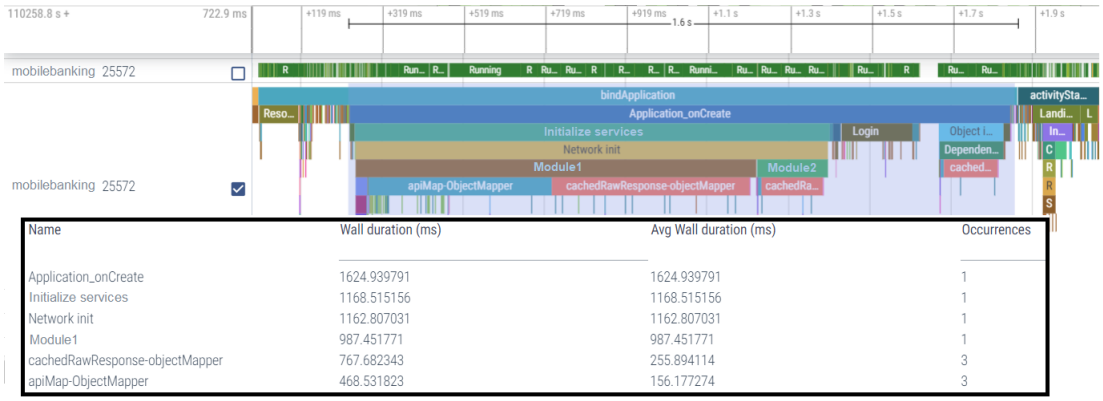
Figure 6.4. Flow chart showing the execution flow of the `invalidate()` method.

The input file contains a JSON string that describes the list of the files

containing the cached responses, and since the number of entries is variable and the mapping between the JSON object and the output class is complex, the first deserialization cannot be avoided.

However, in the right part of the diagram we can observe that a second deserialization operation is performed **for each file** containing a cached response. Similarly to what happened in the case of the 6.1.2, the crucial information about each cached response is stored in one field of the JSON object, and in this case it is the time it was received. If the response has been cached for too long, it must be deleted and a new request will be issued to update it.

To better track the execution of `Application.onCreate()` method, multiple traces have been added inside the code. A more detailed stack is represented in the figure below:



Name	Wall duration (ms)	Avg Wall duration (ms)	Occurrences
Application_onCreate	1624.939791	1624.939791	1
Initialize services	1168.515156	1168.515156	1
Network init	1162.807031	1162.807031	1
Module1	987.451771	987.451771	1
cachedRawResponse-objectMapper	767.682343	255.894114	3
apiMap-ObjectMapper	468.531823	156.177274	3

Figure 6.5. `perfetto` trace showing the execution of the Application `onCreate` method.

The `apiMap-objectMapper` sections correspond to the first deserialization reported in the flow chart, to obtain the list of the cached responses. The `cachedRawResponses` sections, instead, refer to the second deserialization, performed to check the expiry date. Since the impact the `invalidate()` function is significant (especially considering the repeated invocations) a plausible approach to reduce its overhead will be presented in section 6.2.2.

Get user information

In figure 6.5 it is possible to observe a consistent section named `Login`, using the Android Studio CPU profiler we can gain more insights on the operations performed in this part of the trace.

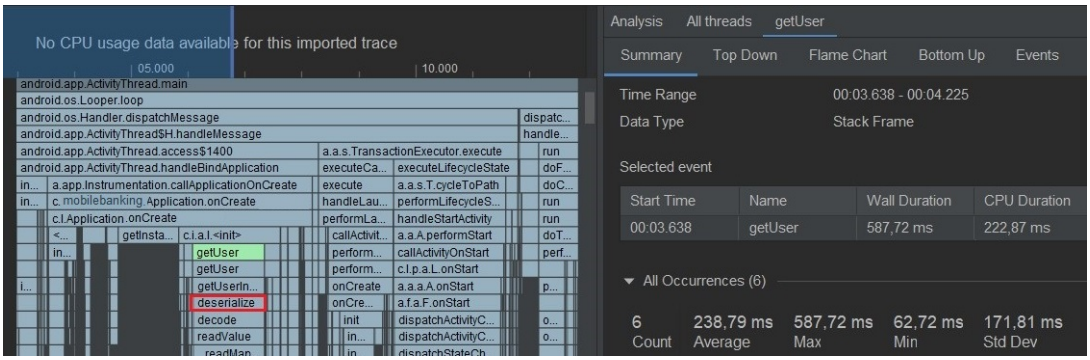


Figure 6.6. `getUser` details in the Android Studio profiler.

Observing the call chart reported in the figure above, we can detect that the `Login` section embeds a method called `getUser` which internally performs a deserialization to convert the JSON file into a `User` class object. Moreover, on the right part of the interface are reported the details for the `getUser` method, and one can notice that not only it is invoked 6 times upon application startup, but the average duration of this operation is significant (238 ms).

Analyzing in more details detail the trace in the CPU profiler, we can better understand the timing of the invocations of the `getUser` method. An interesting detail can be observed in the screenshot below:

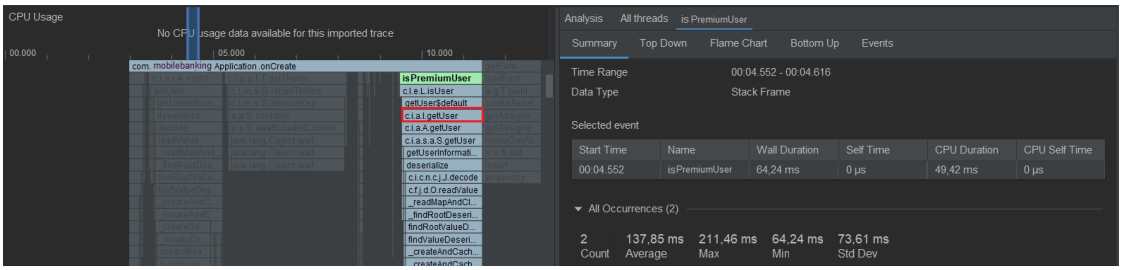


Figure 6.7. `isPremiumUser` details in the Android Studio profiler.

The call stack represented in this section of the trace highlights that the `getUser` method (and the internal deserialization operation) is internally invoked by a function named `isPremiumUser`. As the name suggest, this method's purpose is to check whether the customer account is associated to a "premium" user or not, returning a boolean value (premium users can leverage more functionalities with respect to "normal" users).

The odd thing here is that, similarly to what previously observed in the case of the `Savings` and `CacheHandler` section 6.1.2 and 6.1.2, the heavy deserialization which is internally executed is only aimed at checking the value of a single field. Moreover, the *Summary* section of the CPU profiler reports that this check is performed twice upon application startup, wasting precious time.

A possible improvement for this operation will be presented in section 6.2.2.

6.2 Proposed solutions

6.2.1 Decryption

As highlighted in section 6.1.1, the decryption operation is performed several times upon startup, but for some of the invocations, the input (and the output) are the same. For this reason, the proposed solution consists in adopting a caching behaviour by storing decryption results for identical data, with the goal of avoiding to perform the same expensive decryption operation shortly after. In the next trace we can observe the result obtained by following this approach:

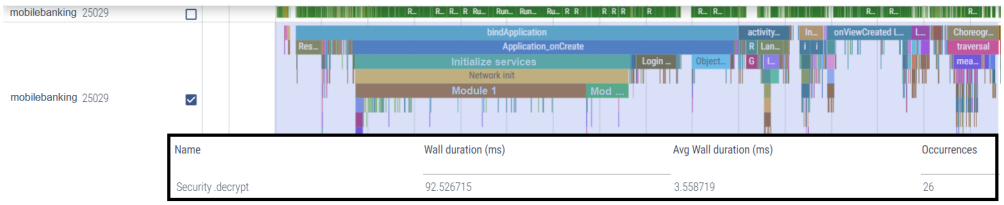


Figure 6.8. Trace showing the time spent for decryption operation when caching the output for identical data.

We can notice that the number of occurrences is the same, as expected, but the total time is noticeably reduced (from 443ms to 92ms).

6.2.2 Deserialization

Activity creation - savings

In section 6.1.2 it was presented that, besides the overhead caused by the inflation of a suitable layout (discussed in section 5.3.1), in order to show the

savings of the customer upon application startup, there was the need of deserializing the information about the user account. By tracing the execution with the debugger, it was possible to identify the order of the instructions performed, which is reported in figure 6.9:

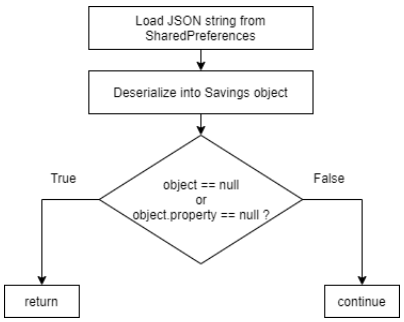


Figure 6.9. Diagram reporting the execution flow for the money-box deserializaion.

It is easy to notice that the expensive deserialization operation is executed before checking the two conditions, whose result depends on the JSON object itself. It is possible to verify that: the resulting JSON object (condition 1) and that the property we're interested in (condition 2) are not null without converting the JSON data into a `Savings` object, but directly manipulating the JSON string. This way, the overhead coming from the deserialization could be avoided when not needed, and executed only if the conditions are met.

By observing figure 6.10 and comparing it with figure 6.2 it is clear that the duration of the `onViewCreated` method is considerably reduced when the deserialization is not performed, requiring less than 25% of its original time.

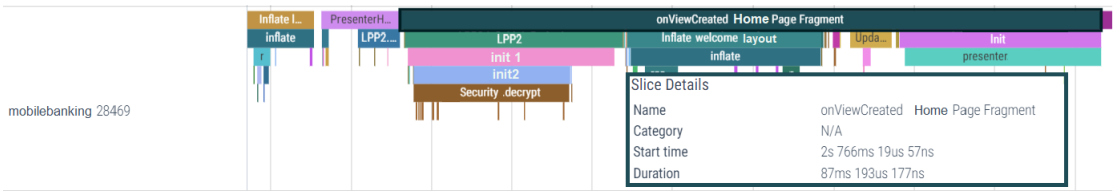


Figure 6.10. Trace showing the impact of avoiding deserialization of the `Savings` object when not needed

Cache Handler

When presenting the `CacheHandler` class in section 6.1.2, it was explained that one of its duties is to check whether cached responses that are stored on the device are out of date, and to do so, multiple deserialization operations are needed. Since the impact on the startup time is remarkable, we can follow a similar approach to what presented in the case of the `Savings` deserialization: the proposed strategy consists in reading the `expiryDate` field from the JSON object and check whether the response file is expired. After that, only valid files will be deserialized and processed. Since the expiry date of the response file is a unique field inside the JSON object, a possible implementation relies on using a **regex**.

The auxiliary function which is in charge of verifying whether the file is too old be preserved in the cache performs the following operations:

1. define regex for the expiry date field:
2. read the response file into a String via the `readText` method;
3. find regex pattern inside the string;
4. extract the value corresponding to the field to obtain the expiry date of the file;
5. compare the expiry date with the policy limit and return *true* if the file is expired, *false* otherwise.

A custom trace, named `readFile` has been defined to encapsulate points 2-4, in order to keep track of the amount of time required to perform these operation and compare it with the previous approach. The result obtained is reported in figure 6.11:

Comparing it with the trace reported in figure 6.5, we can notice that the time required to verify whether the response files must be deleted or not is approximately 4 ms, and it can be considered negligible if compared to the initial duration of such operation (767 ms).

However, the heavy deserialization highlighted in the `apiMap` traces (see figure 6.5) still represents a problem in terms of startup performance. Since the whole `invalidate` operation goal is to find potentially expired cached responses and eventually deleting them from the device memory, a possible improvement would be to adopt a *fire and forget* approach, executing this task on another thread without blocking the main one.

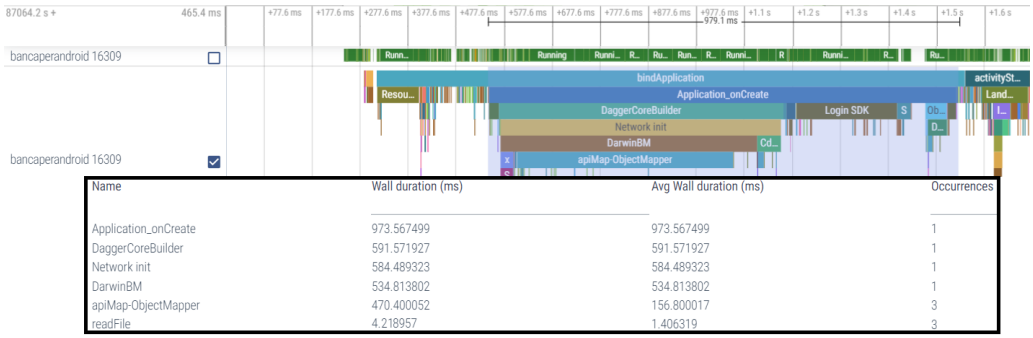


Figure 6.11. `perfetto` trace showing the execution of the `invalidate` method after reading only the expiry date field.

Furthermore, tracing the execution by using the Android Studio debugger, it emerged that the `init` method of the `CacheHandler` class is invoked thrice, each time by a different module of the application. Nonetheless, for each invocation, the only parameter received is the path of the application cache, and it does *not* change for different invocations. This suggests that executing this expensive method only once (for the first module that is initialized) would result in sparing the time needed for the two subsequent invocations.

Even though the possible aforementioned strategies could result in a more efficient initialization of the `CacheHandler` class, their implementation is not easy, since it needs to be shared among different modules of the application. For this reason, a different possible strategy directly affecting the deserialization will be discussed in section 6.2.2.

Get user information

The overhead caused by the repeated calls to the `getUserInformation()` method has been illustrated in section 6.1.2. Recall that this function is invoked several times upon startup, either because the whole `User` object is needed or because the application must only check whether the user is a "premium" one or not.

There are different approaches that can be adopted to mitigate the problem:

- deserialize the JSON string into a `User` object the first time it is needed and cache the obtained value and the starting JSON string. For future requests to retrieve the information about the user, we compare the new JSON string with the cached one and proceed to deserialize it only if

they are different (meaning that some values may have been updated in the meanwhile).

- define an auxiliary function to extract from the JSON string only the field we are interested in (in this case `userType`). This should reduce the time needed to perform this kind of check.

The result of these modifications is not shown in this paragraph, because a common solution to the overhead of the deserialization operation is going to be presented in the next one.

Changing deserialization approach

In the previous sections it has been presented the problem of heavy deserialization upon startup, highlighting the impact that it has on the application at launch time. In the following `perfetto` screenshot, it is possible to observe the total time needed to perform deserialization-related operations when the application is launched in a **COLD state** (N.B. the trace has been recorded after having applied the modifications proposed in section 6.2.2):

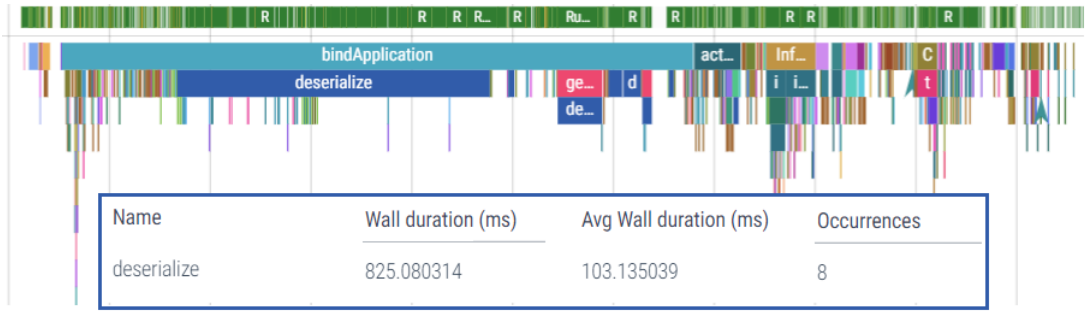


Figure 6.12. `perfetto` trace capturing the aggregated duration of deserialization operations after applying the modifications suggested in section 6.2.2.

Since from the `perfetto UI` we don't have much information on the internal operations performed for this purpose, we can use the Android Studio CPU profiler to dive deeper into the call stack of a deserialization:

Image 6.13 captures the call stack when the `invalidateExpiredCache` method (presented in section 6.1.2) is invoked. We can notice that the library on which the deserialization relies is the `jackson.databind`[1]. Additionally,

[1]More details on the library can be found at the page of its Git repository.

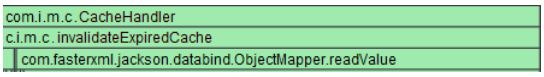


Figure 6.13. Call chart showing the jackson module used for deserialization.

the mapping between the input file (in JSON or XML format, generally) and the output object class is obtained by calling the `readValue()` method of the `ObjectMapper` class.

By reading the online documentation for the **jackson** library, it was possible to discover that there are multiple approaches to deserialize a resource (JSON/XML) into an object: a highly efficient one is the **Streaming API** one[1]. This API offers very good performance in terms of memory and processing overhead, due to the low-level customizable implementation of the deserialization strategy. In fact ,the developer can process the input resource to be parsed one field at a time (using the `Parser` class), making it also suitable to **extract JSON fields** without needing to deserialize the whole object: this last feature is the perfect alternative to the "field extraction" strategy used for the deserialization of the `Savings` object (see section 6.2.2) and the cached responses (see section 6.2.2). However, this approach increases the verbosity of the code, because every detail must be specifically handled in code.

A possible compromise to produce a clearer code, without having to analyze each field separately, combines the efficiency of the **Streaming API** and the handy data-binding capabilities of the **ObjectMapper**. Following this idea, illustrated in this article the "hybrid" deserialization strategy has been adopted for the most significant sections of the **Mobile Banking** application. The total time needed to process the same information highlighted in figure 6.12 is reported in the `perfetto UI` screenshot below:

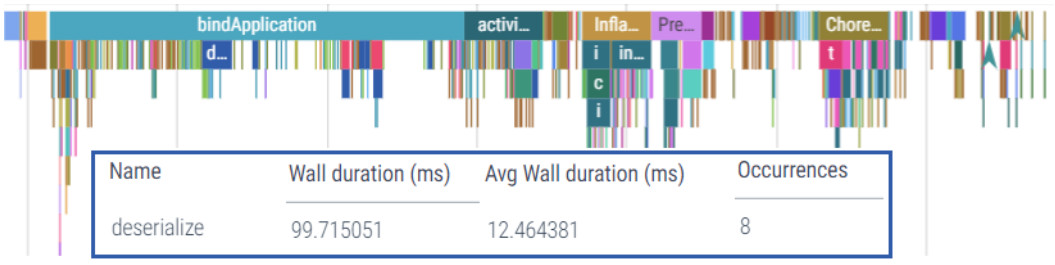


Figure 6.14. `perfetto` trace capturing the deserialization impact after adopting the mixed Streaming API - ObjectMapper approach.

By comparing the total deserialization time required by the initial strategy, reported in figure 6.12, and the one based on the **Streaming API** illustrated in figure 6.14, we can notice that the overall duration of such operation is significantly reduced (from 825 ms to 100 ms). Even though only the main deserialization operations have been modified to adopt the **Streaming API** approach, it can also be extended to less-relevant sections that rely on deserialization.

Chapter 7

Comparison of initial and final launch time of the application

After having presented the critical points for the application startup time in chapters 5 and 6, and implemented most of the proposed solutions in the relative *Proposed solution* section, the goal of this chapter is to compare the startup time observed when considering the application in its initial state and the version in which the suggested modifications have been implemented.

7.1 Data collection

Before presenting the results obtained, it is useful to understand the context of this comparison in terms of devices and versions of the application.

7.1.1 Version of the application

For the analysis of the launch performance and the implementation of the proposed solutions, the **DEBUG** version of the app was utilized. This is due to the fact that, when considering a *debuggable* app, the developer can rely on several instruments that simplify the performance analysis, like:

- the **debugger**: a debuggable version of the application allows to use a debugger (like the one integrated in Android Studio) to track the execution flow of the Android process, gaining information on the current

instruction that is being executed, the value of variables, and the thread's stack. Details on the Android Studio debugger can be found at the following link;

- the **Android Studio CPU profiler**: in section 3.1 have been illustrated the details of such tool, and we have seen that it has been crucial for the analysis of the application;
- the **Layout Inspector**, extremely useful for the layout analysis, carried out in chapter 5;
- the possibility of displaying custom section when collecting system traces. It is worth mentioning that, although one can always specify the beginning and the end of a section via the `Trace` class methods, the information about the fragment of execution will be displayed only if the application is debuggable.

On the other hand, it is important to notice that the **DEBUG** version of an application does not reflect the performance of the ones that end-users install on their devices, that is known as **RELEASE** version. The reason is that the **DEBUG** one does not undergo the optimization process that is usually applied to the marketplace versions of the app, which includes[1]:

- **code shrinking:** removes unused classes, fields and methods;
- **resource shrinking:** removes unused resources from the app;
- **obfuscation:** shortens the name of the classes and their members to reduce the size of the application;
- **optimization:** inspects the code and removes unused statements.

The results that will be presented later in this chapter have been collected using the **UAT** (**U**ser **A**cceptance **T**est) version of the application, that is as similar as possible to the official **RELEASE** one: the main difference is that the first one is executed in an environment specifically though for testing purposes (requests/responses are directed to test servers to avoid overloading production ones).

[1] Android Developers guide on code optimization here

7.1.2 Devices

To better understand the impact of the modifications made on the original application code, different devices have been considered. The following table gives an overview of the most important characteristics:

ID	Device name	API	Released	# cores	CPU freq.	RAM
1	Redmi Note 9 Pro	30	2020/1	8	1.9 GHz	6 GB
2	Honor 9 Lite	28	2018/1	8	2.0 Ghz	3 GB
3	Huawei P10 Lite	26	2017/2	8	1.9 GHz	4 GB
4	Huawei P8	23	2015/2	8	1.8 GHz	3 GB
5	Samsung Galaxy S4	21	2013/2	4	1.9 GHz	2 GB

Table 7.1. Table listing the devices used for launch performance collection.

The devices listed in table 7.1 have been sorted in from the most recent to the oldest one, and we can notice that they also present different hardware specifications. These information will be interesting to take into consideration when presenting the results obtained.

7.1.3 Method

To collect the information about the application startup time, the *script* presented in section 3.3.1 has been used. For each device and for each version of the application (**Initial** and **Final**) the loop presented in the script has been executed for 100 iterations. After that, the obtained results were averaged to obtain a more reliable metric. The waiting time between an iteration and the following one has been set to 3 seconds. The application has been always killed before being launched, to ensure that a **cold state** is observed.

It is also worth mentioning that the metrics have been collected trying to isolate the **Mobile Banking** process as much as possible, meaning that no other application were actively running when measuring the launch times.

7.2 Results

In table 7.2 are reported the results obtained when collecting the application startup time on the devices presented in table 7.1:

	Initial		Final			
Device ID	Time	Std. dev.	Time	Std. dev.	Gain	% Gain
1	1587.38	24.83	881.76	22.87	749.32	47.20
2	2444.11	24.82	1487.36	16.47	956.74	39.14
3	5129.36	276.72	2106.47	22.14	3022.89	58.93
4	9793.62	1108.47	4649.48	47.21	5144.14	52.53
5	8971.01	213.59	4870.91	212.01	4100.1	45.70

Table 7.2. Table reporting the initial and final launch times statistics for different devices. The values are expressed in milliseconds.

From table 7.2 we can notice a clear performance degradation from newer devices to older ones, as one would expect. In fact, the time required by the least recent device (Samsung Galaxy S4, released in 2013) to complete the application startup is roughly six times larger than the one needed to launch the application in the case of the most modern one (Redmi Note 9 Pro, released in 2020). The only exception is observed when considering devices 4 and 5 in the context of the initial version, whose launch times are comparable.

Furthermore, the standard deviation reported for "fresher" devices when executing the **initial** code is substantially smaller than the one observed in older smartphones, suggesting that the performance of such devices can vary more between one execution and another one. Nonetheless, this variability in the launch times seems to decrease when considering the **final** version of the application, leading to more "stable" results: the reduced number of long-lasting operations, that can have different duration from one execution to another contributes to reduce the standard deviation for the observed launch times.

Moreover, comparing the **initial** and **final** launch times we can notice that the improvement is consistent across the different devices, showing a greater absolute gain for older devices, perfectly in line with what expected. Nevertheless, if we take into consideration the percentage gain $(t_{initial} - t_{final})/t_{initial}$, we can observe a considerable percentage improvement also in the case of more recent smartphones, meaning that the modifications

brought to the initial code base are beneficial both for more performing and for less powerful devices.

It is also worth noting that time **initial** startup time registered for devices 3,4 and 5 was above the 5s time limit for a **slow startup** suggested by **Android Vitals** (see section 3.5), with older devices (4 and 5) greatly exceeding it. On the other hand, considering the **final** version of the application, it is possible to observe that the average startup time for the same devices is below this threshold Still, in the case of device number 5 (Samsung Galaxy s4) the results oscillate around this limit.

Chapter 8

Conclusions and future directions

8.1 Conclusions

This work aimed to carry out an analysis of the operations that influence the most the startup time of the **Mobile Banking** application. Employing some of the tools designed for performance analysis, it has been possible to break down the execution of the application at launch time and identify the critical sections that require more time required to be completed and go against the guidelines suggested by the Android team.

Subsequently, a viable solution to reduce the impact of the identified bottlenecks has been proposed, highlighting the benefits that would derive from applying the modifications.

After presenting the most significant issues at the application launch time and the relative suggested modifications, a comparison of the initial and final application startup time was conducted, considering devices with different computational power. The results obtained through this analysis showed that it is possible to improve the startup performance of the application by adopting the strategies presented to reduce the overhead of the critical sections, thus decreasing the time required to launch the application. The proposed approach is beneficial not only for less-performing devices but also for more recent and powerful ones.

However, although the results obtained are promising (see table 7.2 for details), from the user's perspective, improving the startup time from 1.6s to 900ms is less relevant than observing a reduction of the launch time from

9s to 5s. Moreover, even though the percentage gain observed in low-end devices is roughly 50%, a launch time that oscillates around the 5s threshold (above which it is considered a slow start, according to Android Vitals) is likely to be still perceived as "slow", as "61% of users expect mobile apps to load in 4s or less" [11].

A possible solution could be adopting a lightweight version of the application for less recent devices, which provides a reduced number of features compared to the original application, but results in a less complex implementation and therefore decreases the startup time.

Another factor to consider is that the OS API levels of devices for which the absolute gain is greater (26, 23 and 21), are also the least diffused ones, with a total share of approximately 10% (see table 1.1 for details). Even if, as introduced in section 1.1.1, the API level is not necessarily related to the computational power, it can give an estimate devices' recentness. From table 1.1 we can notice the latest 3 OS versions account for approximately 88% of total devices on which the application is installed, and we have seen from table 7.2 that, generally, fresher devices perform better than older ones.

Finally, the effort needed to carry out this kind of analysis must be taken into account. The solutions proposed in this work have been deemed appropriate because optimizing the corresponding critical sections did not require excessive modifications to the original code while still leading to a reasonably positive impact over the startup time. However, the process of identifying the different critical sections and proposing a plausible optimization approach is undoubtedly time-consuming; therefore, it would require a considerable amount of time to perform this work on the complete application.

Nevertheless, the performance analysis of the application should be performed hand in hand with the development of new features, which would also allow to reduce the scope of the study and ease the identification of performance problems.

8.2 Future directions

The analysis conducted in this work is intended to represent a starting point for a more in-depth inspection of the application performance. In fact, even though the application startup surely plays a major role for the customers' satisfaction, the same approach followed in this thesis can be extended to other parts of the application, possibly considering different aspects of the

performance (like network requests, battery consumption, memory management), that collectively contribute to the quality of the product.

Nonetheless, the data collected through this work and the results obtained will be presented to the **Mobile Banking** company, which will evaluate whether it is possible to apply the proposed modifications to the code base or not. Furthermore, this work proposes a methodology for analyzing the performance of an application at startup time (that can also be extended to other parts of the application) and detecting the most critical points, allowing the company to avoid porting the same known issues on future projects.

Bibliography

[1] Baeldung. *Jackson Streaming API*. https://www.baeldung.com/jackson-streaming-api. Sept. 2020. (Visited on 10/11/2021).

[2] Statista Research Department. *Percentage of mobile apps that have been used only once from 2010 to 2019*. https://www.statista.com/statistics/271628/percentage-of-apps-used-once-in-the-us/. July 2021. (Visited on 10/13/2021).

[3] Android Developers. *App Startup Time*. https://developer.android.com/topic/performance/vitals/launch-time. Mar. 2021. (Visited on 10/06/2021).

[4] Android Developers. *Overview of memory management - Share memory*. https://developer.android.com/topic/performance/memory-overview#SharingRAM. Feb. 2021. (Visited on 10/07/2021).

[5] Android Developers. *Overview of System Tracing*. https://developer.android.com/topic/performance/tracing. Oct. 2021. (Visited on 10/06/2021).

[6] Android Developers. *Performance and view hierarchies*. https://developer.android.com/topic/performance/rendering/optimizing-view-hierarchies#cheaper. 2021. (Visited on 10/13/2021).

[7] Android Developers. *Performance and view hierarchies - Double taxation*. https://developer.android.com/topic/performance/rendering/optimizing-view-hierarchies#double. Feb. 2021. (Visited on 10/07/2021).

[8] Android Developers. *ViewStub Documentation*. https://developer.android.com/reference/android/view/ViewStub. 2021. (Visited on 10/21/2021).

[9] Google. *Monitor your app's technical performance with Android vitals*. https://support.google.com/googleplay/android-developer/answer/9844486?visit_id=637692106150122338-3334721575&rd=1#zippy=%2Capp-start-up-time. 2021. (Visited on 10/06/2021).

[10] Joel Newman. *Google I/0 2018'*. https://www.youtube.com/watch?v=dx6LBaFqEHU&t=88s&ab_channel=AndroidDevelopers. May 2018. (Visited on 10/03/2021).

[11] Dimensional Research. *Failing to meet mobile app user expectations. A mobile app user survey*. 2015.

[12] Wikipedia. *Android version history*. https://en.wikipedia.org/wiki/Android_version_history. Oct. 2021. (Visited on 10/12/2021).