

Chapter 11. Performance

Considerations with Android Profiling Tools

Using proficient concurrency in Android leads to better performance in your application. This is why we have made Kotlin concurrency in Android the primary focus of this book. In order to provide a solution for performance bottlenecks, you have to be able to spot them in the first place. Have no worry: this chapter looks at popular Android tooling commonly used to check for potential problems in performance.

Out in the wild, Android faces real-life challenges that affect performance and battery life. For example, not everyone has unlimited data in their mobile plans, or reliable connectivity. The reality is that Android apps must compete with one another for limited resources. Performance should be a serious consideration for any Android application. Android development doesn't stop at creating an app. Effective development also ensures a smooth and seamless user experience. Even if you have a deep understanding of Android development, your application may have issues such as:

- Decrease in performance
- Slow startup/slow response to user interactions
- Battery drain
- Wasteful use of resources, and clogged memory
- UI bugs that don't force a crash or generate an exception, but nevertheless affect user experience

This list of sudden, strange behaviors in an app is by no means exhaustive. As previous chapters showed, managing multithreading can become complex when there are also interacting Android components to keep track of. Even if you have a solid understanding of multithreading, it's hard to say how an application really works until we analyze performance with profiling tools. To answer these kinds of ambiguities, there are several useful tools for profiling various aspects of Android. Four of them can be retrieved and used right in Android Studio, as diagrammed in [Figure 11-1](#).

In this chapter, we look at profiling tools in Android Studio’s *Android Profiler* and a popular open source library called *LeakCanary*. We explore each one by profiling a real-life application for potential performance bottlenecks. Remember the hiking application described in previous chapters? Surprise! It was inspired by *TrekMe*. TrekMe is an Android trail-trekking app, an open source Android project where users download interactive topographical hiking routes to use offline later while on hikes. TrekMe started as a Java project, but its codebase is currently 80%+ Kotlin. Here are some important features of TrekMe that users of the application can enjoy:

- Download topographical maps for offline use.
- Get the device’s live position even when there’s no network, while the app tries its best to preserve battery life.
- Track hikes in great detail without draining the device’s battery when you need it most.
- Access other useful information without needing an internet connection (save for creating the map).

We encourage you to explore TrekMe so you can follow along with this chapter. You can [retrieve the source code from GitHub](#). Once you’ve cloned the project, open it with Android Studio. Finally, run an instance of an emulator from the *Android Virtual Device (AVD) Manager* that you intend to run TrekMe on.


Performance considerations are crucial. It’s not uncommon to find performance lag in any application, but such a “fishing expedition” must be approached with care. It’s up to the developer to decide on the most relevant tooling, and which optimizations outweigh in benefits the cost of their creation. Profiling your app allows you to investigate application performance objectively. To give some examples of the kinds of surprises you might encounter, we’ll look at TrekMe with Android Profiler.

Android Profiler

Android Profiler analyzes an application’s session to generate real-time feeds for CPU usage and memory usage, as well as network and energy profiling. [Figure 11-2](#) shows Android Studio with the TrekMe application runtime showing in the bottom half of the console.

Figure 11-2. A profiling session records profiling data. The active session attaches to the running app in the emulator (not pictured).

Android profiling can be instantiated in three ways:

1. If your application is not running, click the Profile app icon in the upper-right corner to instantiate the app and the profiler at once. This action builds and compiles a new running instance of the application. Android Studio will then open a new session giving you a stream of your data in real time.
2. If your application is already running, click the + icon and select the running emulator.
3. You can also import a previously saved profiling session with the  icon. From there, you can load the previously saved *.hprof* file.

You can record and store data in each session. In [Figure 11-3](#), we show a screenshot of saved profiling sessions with different kinds of data that can be recorded with Android Profiler.

Figure 11-3. Save heap dumps, or different kinds of CPU traces.

Both *method traces* and *heap dumps* can be saved as separate entries within a running session. Method traces show a stacktrace of methods and functions that can be recorded in CPU profiling. Meanwhile, a heap dump refers to the data collected from *garbage collection*, allowing us to analyze what objects are taking up unnecessary space in memory.

Android Profiler records one application session at a time. However, you can save multiple recordings and switch between them to compare the data. A bright dot indicates the recording of an active session. In [Figure 11-3](#), there are three recorded sessions. The last recorded session has a saved heap dump, which refers to a log of stored memory in the JVM at the time of the snapshot. We'll cover this in more detail in [“Memory Profiler”](#). The first recorded session saved different kinds of CPU recordings. This will be discussed in [“CPU Profiler”](#).

NOTE

Android Studio caches sessions only for the lifetime of the Android Studio instance. If Android Studio is restarted, the recorded sessions will not save.

The following sections show in more detail how Android Profiler evaluates device resources in the virtual machine at runtime. There are four profilers we'll use: *Network Profiler*, *CPU Profiler*, *Energy Profiler*, and *Memory Profiler*. All of these profilers record streams of data during an application's runtime, which can be accessed in greater detail in their own special views.

By design, TrekMe encourages users to download detailed topographical maps directly to their devices while they're at home and can do so easily. Creating new topographical maps in TrekMe is the feature that consumes the most resources in this process. The maps can then be rendered when the user is hiking, even if mobile coverage is unreliable. TrekMe's map creation feature allows you to select an official map generator like the *Instituto Geografico Nacional* (IGN) or *U.S. Geological Survey* (USGS) or some other map provider, as shown in [Figure 11-4](#). TrekMe will then load the selected service's map in square tiles, one by one.

Figure 11-4. TrekMe allows you to create and download a map from different services.

For the remainder of this chapter, we'll profile TrekMe while creating a map via IGN to study the time it takes to load a map, and to ensure that it is optimal. With Android profiling, we can explore questions like:

- Are we making fast network calls?
- Is the data we get in our response returned in the most efficient format?
- What parts of the application are the most CPU-intensive?
- Which Android actions drain the most battery?
- What objects are eating up the most memory in heap?
- What consumes the most memory?

In the next section, we answer the first two questions with Network Profiler. We explore the remainder of these questions in later sections.

Network Profiler

When a network call is made, the radio in the Android device powers up to allow for network communication. This radio then stays powered on for a short time to ensure there are no additional requests to listen for. On some phones, using the network every two minutes keeps the device at full power forever. Too many network calls can be expensive for

Android resources, so it is important to analyze and optimize network use in an application.

Network Profiler generates connection breakdowns used by *HttpURLConnection* or *OkHttp* libraries. It can give you information like network request/response time, headers, cookies, data formats, the call stack, and more. When you record a session, Network Profiler generates interactive visual data while you continue to interact with the application.

When we create a map using IGN, TrekMe renders the map on the screen in square tiles, one by one. Sometimes, though, the tile rendering seems to take a long time. [Figure 11-5](#) shows the profiler capturing incoming/outgoing network requests, and shows the connections that are available while creating a map on TrekMe via IGN:

You can highlight a selected range of the timeline to drill into these connections further, which will expand a new view of the Network Profiler workspace, allowing you to access the *Connection View* and *Thread View* tabs to analyze these network calls further.

Figure 11-5. Network Profiler timeline records IGN Spain map creation on TrekMe. In the upper-left corner of the chat, the long line under the label `MainActivity` represents an active `Activity` session while the short, thick line above the `MainActivity` label with a dot at the left represents user touch events.

Viewing network calls with Connection View and Thread View

Connection View shows the data that was sent/received. You can see this in [Figure 11-6](#) in the highlighted portion of the timeline. Perhaps what is most notable is Connection View's ability to sort resource files by size, status, and time. Clicking the header of each section will organize the ordering of the desired filter. The timeline section represents the timing of the request/response bars split into two colors. The lighter portion represents the duration of the request, while the darker portion represents the duration of the response.

Figure 11-6. Connection View shows a list of individual network calls.

Connection View looks similar to the timeline in Thread View, but they're not quite the same. Thread View shows the network calls being made within the designated initiating threads, which can show multiple net-

work calls running in parallel time. The screenshot shown in [Figure 11-7](#) is the complement of the previous image, using the same data set.

Figure 11-7. Thread View shows a list of network calls made within each thread.

Seeing how worker threads divide labor in real time can help to reveal areas for improvement. TrekMe's pooled threads are responsible for automatically breaking up, as needed, the work of downloading all these images.

Both images show roughly 23 seconds of network calls, with response times showing a similar trend. Compared to the requests, responses appear to take up a disproportionate amount of the time it takes to complete an entire network call. There could be several reasons for this: for example, the server connection might be weaker if a device attempts to pull this data from a distant country. Perhaps there are inefficiencies with the query call in the backend. Regardless of the reason, we can say that our network calls may not be fastest. However, the presence of fast request times *and* slow response times indicates external factors that are out of the device's control.

We now turn to our second question: are we using the most efficient data format? Let's look at the connection type in the Connection View tab as pictured in [Figure 11-6](#). If you don't need transparency in your images, avoid using PNG files since the file format doesn't compress as well as JPEG or WebP. In our case, the network calls return a JPEG-formatted payload. We want files that provide consistent and good image quality to enable users to zoom in to the details of those images as much as they need to. Using a JPEG file also takes up less memory than a PNG file would.

We can get more granular detail on each network call and its payload by selecting any item: this opens a new view within Network Profiler on the right side, showing tabs for Overview, Response, Request, and Callstack. In the next section, we'll be able to look into the specifics of a single network call and locate where the network call is made in the code.

Network call, expanded: Overview | Response | Request | Callstack

Android developers are used to working with other platforms in order to achieve feature parity and more. Suppose a network call starts returning the wrong kind of information for a network request. The API team is in

need of specifics for the network request and response you're getting on the client side. How can you send them over the necessary request parameters and content headers they need to investigate on their side?

Network Profiler gives us the ability to inspect network responses and requests on the right-side panel in Connection View or Thread View, as shown in [Figure 11-8](#).

The *Overview* tab details notable highlights captured in the request and response:

Request

The path and potential query parameters

Status

The HTTP status code returned within the resulting response

Method

The type of method used

Content type

The media type of the resource

Size

The size of the resource returned in the resulting response

Figure 11-8. Network Profiler allows you to inspect response and request information.

The *Request* and *Response* tabs show a breakdown of headers, parameters, body data, etc. In [Figure 11-9](#), we show the exact network call as in the previous image, except with the *Response* tab selected.

As you can see in the network response, TrekMe uses a basic HTTP API. Other types of API data formats return HTML, JSON, and other resources. When applicable, the *Request* and *Response* tabs offer body data as a formatted or raw representation. In our case, the resource media returns JPEGs.

Figure 11-9. Network Profiler captures network calls to render map.

Finally, the *Call Stack* tab, shows the stacktrace for the relevant calls made to execute a network connection, as pictured in [Figure 11-10](#). The calls that are not faded represent the method calls within the call stack coming from your own code. You can right-click the calls indicated to be able to jump to the source code with ease.

Network Profiler is useful for more than just analytics. As you can see for yourself, you're able to process a lot of information quickly. From caching repetitive calls to confirming API contracts, Network Profiler is a tool worth keeping in your toolbox.

Figure 11-10. Call Stack tab.

Poor networking is not the only culprit when it comes to slow rendering times. The task of creating a brand new topographical map is heavy in itself, but as we have determined from a networking stance, no further action is required to improve loading times or data format. However, we would be remiss to chalk up slow loading times to slow response time alone. After TrekMe receives the network data, it must then process the data to render the UI. For this reason, we should check for potential inefficiencies in drawing the map out after the network calls. *CPU Profiler* is able to provide insight for this. In the next section, we will examine, using CPU Profiler, the processing consumption of the rendering of the IGN Spain map.

CPU Profiler

While Network Profiler is able to give information about network calls, it is not able to paint a full picture about where the time goes. We have a call stack for our network calls, but we don't know how long certain methods actually run. This is where CPU Profiler comes in. CPU Profiler helps identify greedy consumption of resources by analyzing how much time has passed on function execution and tracks which thread a call executes on. Why does this matter? If TrekMe consumes too much processing, the application slows down, impacting the user experience. The more CPU power that is used, the more quickly the battery drains.

CPU Profiler allows you to examine CPU recordings and livestream data by examining the call stack by the thread, as shown in [Figure 11-11](#).

In the following sections, we break down the CPU timeline, Thread activity timeline, and Analysis panels. Because TrekMe seems to spend a lot of

time offloading work to background threads, we will select one to look into more closely.

Figure 11-11. CPU Profiler shows the call stack and recorded times for methods executed.

CPU timeline

The CPU timeline organizes regional call stacks into recorded threads in the Threads pane. The graph in [Figure 11-12](#) shows spikes of CPU usage, where the number is a percentage of available CPU. If you have made a trace recording, you should be able to highlight the CPU timeline to see more information.

Figure 11-12. CPU timeline.

Android Studio allows you to drag-and-click over a recorded sample from the CPU timeline to show the Call Chart. Clicking on Record brings you to a separate trace CPU recording screen (covered in greater detail in *Record Traces*). To create the more granular call charts we explore in the next section, it helps to highlight smaller portions of the recorded CPU trace.

Thread activity timeline

The Thread activity timeline accompanies the CPU timeline showing every running thread in the app. If a section was trace-recorded, you should be able to select a thread to view the call stack captured within the selected time range. In [Figure 11-13](#), 31 threads have been created and used within the application. These threads have been created either by your code, the Android OS, or a third-party library.

Figure 11-13. Thread activity timeline.

The lightest-colored blocks represent a running or active thread. There's not a lot to see on the Main thread, but remember, this image captures a CPU trace of the network request downloading the map images. In this case, we expect background threads to do the necessary work to download the network data. It seems we have the main thread waiting on one of the DefaultDispatcher threads for half the time. Double-clicking on an individual thread expands the call stack.

Below the Thread activity timeline is the Call Chart (see [Figure 11-14](#)).

Figure 11-14. The Call Chart shows a top-down representation of captured methods.

The Call Chart shows a call stack of the segmented range of time for CPU usage. The top boxes represent the encapsulating parent method, while the methods below are child methods that were called. The parent method waits on the child methods to finish executing, so this is a good place to see which of TrekMe's methods could be executing for a long time, like the method `TileStreamProviderHttp`.

If you're reading the printed book, be aware that the bars are color coded. Android OS methods are orange, methods you've written are green, and third-party libraries are blue. Within this coroutine, the longest amount of execution time is with

`TileStreamProviderHttp.getTileStream(...)`. This is expected, given that this call makes individual network requests per tile.

Analysis panel

The *Analysis panel* presents a layered tab view. The top of the pane highlights the active set of thread(s). Beneath the tabbed menu sits a search bar above the stacktrace. You can use the search bar to filter trace data related to a particular call. Below that is a set of tabs intended to render visual data from method tracing in three views: *Top Down*, *Bottom Up*, and *Flame Chart*.

Top Down renders a graphical representation of method traces from the top to the bottom of the chart. Any call made within a method renders as a child underneath the original method. Shown in [Figure 11-15](#), the method `getTileStream` used in TrekMe waits for a series of calls for internet connection and reading from a data stream.

The Top Down view shows how CPU time breaks down in three ways:

Self

The method execution time itself

Children

The time it takes to execute callee methods

Total

Combined time of self and children

Figure 11-15. Top Down view.

In the case of `getTileStream`, the majority of the time is spent on the network calls themselves: in particular, the connection request and `getInputStream` to receive incoming data from the network. For the IGN Spain server, these times can vary when accessed in another country and at different times of the day. Because it is the client consuming server data, TrekMe has no control over how the server performs.

Contrary to Top Down, Bottom Up (shown in [Figure 11-16](#)) shows an inverse representation of *leaf elements* of the call stack. In comparison, such a view renders a substantial number of methods, which can be useful in identifying methods that are consuming the most CPU time.

The final tab provides a Flame Chart view. A Flame Chart provides an aggregated visual of operations from the bottom up. It provides an inverted call chart to better see which functions/methods are consuming more CPU time.

Figure 11-16. Bottom Up view.

To summarize, CPU profiling can render three different kinds of views, depending on the kind of deep dive you wish to pursue:

- Top Down graphical representation shows each method call's CPU time along with the time of its callees.
- Bottom Up inverts the Top Down representation and is most useful to sort methods consuming the most or the least amount of time.
- The Flame Chart inverts and aggregates the call stack horizontally with other callees of the same level to show which ones consume the most CPU time first.

Not only are there three different ways to render data, but there are different kinds of call stacks you can record. In the upcoming sections, we cover different kinds of method tracing in CPU Profiler. As you're starting to get the picture of what kind of information CPU Profiler tries to capture, we'll turn to *method tracing* with CPU Profiler and record a segment of TrekMe creating a new map.

Method tracing

CPU Profiler allows you to *record a trace* to analyze and render its status, duration, type, and more. Tracing relates to recording device activity over a short period of time. Method tracing doesn't occur until the recording button is clicked twice: once to start the recording, and another time to end the recording. There are four configurations for samples and traces, as shown in [Figure 11-17](#).

Figure 11-17. Configurations are available for Android developers for samples and traces.

Sample Java Methods captures the application call stack, or a Call Chart (also seen in previous sections). The Call Chart renders under the Thread activity timeline, which shows which threads are active at a particular time. These traces store individual sessions to the right pane for comparison with others' saved sessions.

By choosing the Sample Java Methods configuration, you can examine TrekMe's call stack by hovering the mouse pointer over particular methods, as shown in [Figure 11-18](#).

Figure 11-18. Sample Java Methods.

WARNING

Don't let your recording run too long. Once a recording reaches its size limit, the trace stops collecting data even if the current session continues to record.

Unlike Sample Java Methods, *Trace Java Methods* strings together a series of timestamps recorded for the start and end of a method call. Should you wish, you can monitor *Sample C/C+ Functions* to gain insight into how the app is interacting with the Android OS. Recording sample traces for native threads is available for Android API 26 and up.

The terms “method” and “function” tend to be used in everyday conversation interchangeably when talking about method-tracing analysis. At this point, you might be wondering why Java methods and C/C++ functions differentiate enough to matter in CPU profiling.

In the CPU-recording configurations, Android Profiler uses “method” to refer to Java-based code, while “function” references threads. The difference between the two is the order of method execution preserved via a

call stack while threads are created and scheduled by the Android OS itself.

Finally, there is Trace System Calls in the configurations shown in [Figure 11-17](#). System Trace is a powerful CPU-recording configuration made available for Android developers. It gives back graphical information on frame-rendering data.

Trace System Calls records analytics on *CPU Cores* to see how scheduling occurs across the board. This configuration becomes more meaningful for detecting CPU bottlenecks across the CPU Cores. These kinds of bottlenecks can jump out in places where the *RenderThread* chokes, especially for red-colored frames. Unlike other configurations, Trace System Calls shows thread states and the CPU core it currently runs on, as shown in [Figure 11-19](#).

One of the key features in a system trace is having access to the *RenderThread*. *RenderThread* can show where performance bottlenecks might be occurring when rendering the UI. In the case of [Figure 11-19](#), we can see that much of the idle time occurs around the actual drawing of the tiles themselves.

The Android system tries to redraw the screen depending on the refresh rate on the screen (between 8 ms and 16 ms). Work packets taking longer than the frame rate can cause *dropped frames*, indicated by red slots in Frames. Frames drop when some task does not return before the screen redraws itself. In the case of this system trace recording, it appears that we indeed have some dropped frames indicated by the numbers labeling boxes inside the Frame subsection under the Display section.

TrekMe saves each frame into a JPEG file and loads the image into a bitmap for decoding. However, in [Figure 11-19](#), we see that in the *RenderThread*, the length of *DrawFrame* doesn't quite match up with the draw rate intervals. A bit farther below that, some of that idle time is tied to various long-running `decodeBitmap` methods in the pooled threads.

Figure 11-19. System Trace reveals dropped frames where times are labeled within Frames.

From here, there are some options that could potentially be considered for faster drawing; that is, caching network responses for images, or even *prefetching*. For users in need of a few megabytes of data, prefetching is a nice-to-have in the case a device has access to at least a 3G network. The

problem with that is that it may not be the best option to render those bitmaps before we *know* what must be rendered. Another option is potentially encoding the data into a more compressed format for easier decoding. Whatever the decision, it's up to the developer to evaluate the trade-offs and the effort of implementing certain optimizations.

NOTE

The concept of prefetching refers to predicting what kind of data would come in a future request, and grabbing that data preemptively while there's an active radio connection. Each radio request has overhead in terms of the time it takes to wake up the radio and the battery drainage that occurs to keep the radio awake, so Android developers can take advantage of making additional calls while the radio is already awake.

Recording a sample method trace

Now that you are more familiar with what the recording configurations offer, we turn to *Sample Method Trace* on TrekMe. CPU recordings are separated from the CPU Profiler timeline. To begin, click the Record button at the top of the screen to analyze CPU activity while interacting with TrekMe.

Ending the recording renders a tabbed right pane of execution times for sample or trace calls. You can also highlight multiple threads at once for analysis. The average Android developer may not use all these tabs all the time; still, it's good to be cognizant of what tools are at your disposal.

In TrekMe, there's a predefined set of iterable tiles to download. A number of coroutines concurrently read the iterable and perform a network request per tile. Each coroutine decodes a bitmap right after the network request succeeded. These coroutines are sent to some dispatcher such as `Dispatchers.IO`, and the rendering happens when the result is sent back to the UI thread. The UI thread is never blocked waiting for bitmap decoding, or waiting for a network request.

The shrunken CPU timeline in [Figure 11-20](#), at first glance, appears to be nothing more than a reference to the previous screen view. However, you can interact with this data to drill down further by highlighting a chunk of time via the range selector, as shown in [Figure 11-21](#).

Figure 11-21. The range selector helps to manage sections of highlighted ranges.

In [Figure 11-22](#), we look at one of the longer-running methods, `getTileStream`. Below the timeline, the left panel allows you to organize *threads* and *interactions* via drag-and-drop functionality. Being able to group threads together also means you can highlight groups of stack-traces. You can expand a thread in a recorded trace by double-clicking the thread twice to show a drop-down visual of a call stack.

Selecting an item also opens an additional pane to the right. This is the *Analysis Panel*, which allows you to examine stacktrace and execution time in more granular detail. Tracking CPU usage is important, but perhaps you'd like to be able to analyze how an application interacts with Android hardware components. In the next section, we look into Android Studio's *Energy Profiler*.

Figure 11-22. You can search for a specific method via the search function.

Excessive networking calls on Android devices are also *power-hungry*. The longer the device radio stays awake for network communication, the more CPU consumption and battery drainage there is. By this logic, it would be fair to assume that networking accounts for most energy consumption. We can confirm this by using Energy Profiler.

Energy Profiler

Energy Profiler is best used for determining heavy energy consumption. When an application makes a network request, the application turns on the mobile radio hardware component. CPU consumption accelerates as the Android device communicates with the network, draining battery at a faster rate.

TrekMe prescales bitmaps to ensure consistent memory and energy usage when the user is zooming in and out. When the user is creating and downloading a map, the details of the map are, by default, downloaded with the highest-resolution detail. The event pane shows higher levels of consumption when downloading large chunks of data.

A drag-and-click can select a range of the timeline to show details for events for the Android OS. In [Figure 11-23](#), we can see a pop-up rendering of a breakdown of the energy graph. The first half of the pop-up legend

contains the categories CPU, Network, and Location, which relay to each category provided in the stacked graph. It is a good sign to see that CPU and networking usage is light despite the relatively heavy job of making a network call to request large pieces of data and draw them on the screen.

Figure 11-23. System event pane.

The second half of the pop-up legend describes the kinds of system events captured from the device. Energy Profiler works to capture certain kinds of system events and their energy consumption on a device:

- *Alarms* and *Jobs* are system events designed to wake up a device at a specified time. As a best practice, Android now recommends using *WorkManager* or *JobScheduler* whenever possible, especially for background tasks.
- *Location* requests use Android GPS Sensor, which can consume a large amount of battery. It's a good practice to make sure accuracy and frequency are gauged correctly.

Although [Figure 11-23](#) shows only one location request, there are other types of system events that contain their own unique set of states. A request event may possess the state of *Active*, as pictured in [Figure 11-23](#), *Requested*, or *Request Removed*. Likewise, if Energy Profiler captures a *Wake Lock* type of system event, the timeline would be able to show state(s) for the duration of the wake lock event such as *Acquired*, *Held*, *Released*, and so on.

Selecting a particular system event opens a right pane in Energy Profiler to see more details. From here, you can jump directly to the source code for that particular location request. In TrekMe, `GoogleLocationProvider` is a class that polls for user location every second. This isn't necessarily an issue—the polling is intended to enable the device to constantly update your location. This proves the power of this profiling tool: you can get precise information without looking at the source code. Requests are made one at a time, removing existing requests in order to make a new one when a new image block has been downloaded.

In comparison to location polling, we can expect decreased energy consumption when a user is zooming in on a rendered map. There are no requests made for downloading large chunks of data. We do expect some

energy consumption for keeping track of the user's location, which also uses `GoogleLocationProvider`.

In [Figure 11-24](#), we can see the excessive and rapid touch events indicated by the circular dots above the stacked overlay graph. Because TrekMe has downloaded all the information it needed, no network calls are made at this time. However, we do notice how CPU usage spikes back up to high levels. To avoid overwhelming the system, it is a good practice to limit touch events to avoid spinning off duplicate zoom-drawing functions.

Figure 11-24. TrekMe opens and zooms in on an existing map.

So far, we've covered evaluating performance by looking at processing power. But examining battery/CPU usage does not always diagnose performance problems. Sometimes, slow behavior can be attributed to clogged memory. In the next section, we explore the relationship between CPU and memory and use Memory Profiler on TrekMe's GPX recording feature.

Memory Profiler

In TrekMe, you can navigate to *GPX Record* in the pullout drawer. GPX stands for *GPS Exchange Format* and is a set of data used with XML schema for GPS formatting in software applications. Hikers can click the play icon under Control. The app then tracks and records the movements of the hikers and their devices, which can be saved as a GPX file to be rendered as a line drawing later on to indicate the path traveled. [Figure 11-25](#) shows TrekMe's GPX recording feature.

Figure 11-25. TrekMe's GPX recording feature uses `GpxRecordingService` to track the GPS coordinates of a user on a hike.

We know that using location in the system *can* be heavy for CPU processing. But sometimes, slowdowns can be attributed to memory problems. CPU processing uses RAM as its capacity for workspace, so when RAM fills up, the Android system must execute a heap dump. When memory usage is severely restricted, the ability to execute many tasks at once becomes limited. The more time it takes to execute fewer application operations, the slower Android gets. RAM is shared across all applications: if too many applications are consuming too much memory, it can slow the per-

formance of the device or, worse, cause `OutOfMemoryException` crashes.

Memory Profiler allows you to see how much memory is consumed out of the memory allocated for your application to run. With Memory Profiler, you can manually trigger a heap dump in a running session to generate analysis to determine which objects are held in the heap and how many there are.

As shown in [Figure 11-26](#), Memory Profiler offers powerful features:

- Triggering garbage collection
- Capturing a Java heap dump
- Allocation tracking
- An interactive timeline of the fragments and activities available in the Android application
- User-input events
- Memory count to divide memory into categories

Figure 11-26. Allocation Tracking offers a *Full Italicized Text* configuration, which captures all object allocations in memory, while a *Sampled* configuration records objects at regular intervals.

NOTE

Like recording samples and traces in CPU Profiler, capturing Java heap dumps saves the results within the session panel in Android Profiler for comparison for the life of your Android Studio instance.

Initiating too much garbage collection (GC) can affect performance: for example, executing a ton of GC can slow the device down, depending on how frequent and how large generational object allocation is in memory. At a minimum, Android developers should try to run memory profiling of every application to ensure that nothing is being held in the heap past its use, otherwise known as “memory leaks.” Detecting memory leaks can be life-saving, especially for Android users depending on longer battery life. What you are about to see is a variation of a common memory management mistake developers often make while working with services: leaving a service accidentally running.

TrekMe uses a foreground service to gain stats of the user’s hike, which is a natural choice for tracking the user’s location. Services, like other Android components, run in the UI thread of the application. However,

persisting services tend to drain battery and system resources. Hence, it is important to limit the use of foreground services so as not to impair overall device performance and to kill them off as soon as possible if the app must use one.

We can run a couple of GPX recordings against Memory Profiler and trigger the heap dump to see which objects held in heap consume the most memory, as shown in [Figure 11-27](#).

Figure 11-27. You can use the CTRL + F function to search for “GpxRecordingService” to narrow your results.

A heap dump shows you a list of classes, which can be organized by heap *allocations*, *native size*, *shallow size*, or *retained size*. Shallow size is a reference to the total Java memory used. Native size is a reference to the total memory used in native memory. Retained size is made of both shallow size and retained size (in bytes).

Within a recorded heap dump, you can organize your allocation record by *app heap*, *image heap*, or *zygote heap*. The zygote heap refers to the memory that is allocated for a zygote process, which might include common framework code and resources. The image heap stores memory allocation from the OS itself and contains references to classes used in an image containing our application for a system boot. For our use case, we’re more concerned with the app heap, which is the primary heap the app allocates memory to.

In Memory Profiler, triggering a heap dump will render a list of objects still held in memory after GC. This list can give you:

- Every object instance of a selected object displayed in the *Instance View* pane, with the option to “Jump to Source” in the code
- The ability to examine instance data by right-clicking an object in *References* and selecting *Go to Instance*

Remember, a memory leak occurs when caching holds references to objects that are no longer needed. In [Figure 11-28](#), we search for “Location” with the same heap dump to locate our service and be able to view total memory allocation. `LocationService` appears to have separate allocations when it should only have one running at a time.

Figure 11-28. A suspicious number of `LocationService` instances appears to be held in memory.

It appears that every time we press Record, a new `LocationService` in TrekMe is instantiated and then held in memory even after the service dies. You can start-and-stop a service, but if you are holding a reference to that service in a background thread, even if it is dead, the instance continues to be held in the heap even after GC occurs.

Let's just run a couple more recordings in TrekMe to confirm the behavior we suspect. We can right-click one of these instances to "Jump to Source" and see. In *RecordingViewModel.kt*, we see the following code:

```
fun startRecording() {  
    val intent = Intent(app, LocationServices::class.java)  
    app.startService(intent)  
}
```

We want to check whether these services are indeed stopping before starting a new one. A started service stays alive as long as possible: until a `stopService` call is made outside the service or `stopSelf` is called within the service. This makes the use of persistent services expensive, as Android considers running services always in use, meaning that the memory a service uses up in RAM will never be made available.

When a GPX recording stops, `LocationService` propagates a series of events, pinging the GPS location, which is then recorded and saved as a set of data. When a GPX file has just been written, the service subscribes to the main thread to send a status. Because `LocationService` extends Android `Service`, we can call `Service::stopSelf` to stop the service:

```
@Subscribe(threadMode = ThreadMode.MAIN)  
fun onGpxFileWriteEvent(  
    event: GpxFileWriteEvent  
) {  
    mStarted = false  
    sendStatus()  
    stopSelf()    // <--- fix will stop the service and release the reference  
}
```

We can use Memory Profiler and check the heap dump to ensure we hold reference to only one service in memory. Actually, since GPX recordings are done through `LocationService`, it makes sense to stop the service when the user stops recording. This way, the service can be deallocated from memory on GC: otherwise, the heap continues to hold an instance of `LocationService` past its life.

Memory Profiler can help you detect possible memory leaks through the process of sifting through the heap dump. You can also filter a heap dump by checking the *Activities/Fragments Leaks* box in the heap dump configurations in Memory Profiler. Hunting for memory leaks can be...a manual process, and even then, hunting for memory leaks yourself is only one way of catching them. Luckily, we have LeakCanary, a popular memory leak detection library that can attach to your app in debug mode and idly watch for memory leaks to occur.

Detecting Memory Leaks with LeakCanary

LeakCanary automatically detects at runtime explicit and implicit memory leaks that might be hard to detect manually. This is a great benefit, since Memory Profiler requires manually triggering a heap dump and checking for retained memory. When crash analytics are unable to detect crashes coming from an `OutOfMemoryException`, LeakCanary serves as a viable alternative to keep an eye on issues detected at runtime, and offers better coverage in discovering memory leaks.

Memory leaks commonly come from bugs related to the lifecycle of objects being held past their use. LeakCanary is able to detect various mistakes such as:

- Creating a new `Fragment` instance without destroying the existing version first
- Injecting an Android `Activity` or `Context` reference *implicitly* or *explicitly* into a non-Android component
- Registering a listener, broadcast receiver, or RxJava subscription and not remembering to dispose of the listener/subscriber at the end of the parent lifecycle

For this example, we have installed LeakCanary in TrekMe. LeakCanary is used organically in development until a heap dump with potential leaks has been retained. You can install LeakCanary by adding the following dependency to Gradle:

```
debugImplementation 'com.squareup.leakcanary:leakcanary-android:2.*'
```

Once installed in your application, LeakCanary automatically detects leaks when an `Activity` or `Fragment` has been destroyed, clears the

`ViewModel`, and more. It does this by detecting retained objects passed through some `ObjectWatcher`. LeakCanary then dumps the heap, analyzes the heap, and categorizes those leaks for easy consumption. After installing LeakCanary, you can use the application like normal. Should LeakCanary detect retained instances in a heap dump that occurs, it sends a notification to the system tray.

In the case of TrekMe, it appears LeakCanary has detected a memory leak within a `RecyclerView` instance of `MapImportFragment`, as shown in [Figure 11-29](#).

Figure 11-29. LeakCanary shows a `RecyclerView` leaking in its stacktrace.

The error message is telling us that a `RecyclerView` instance is “leaking.” LeakCanary indicates that this view instance holds a reference on a `Context` instance which wraps the activity. Something prevents the `RecyclerView` instance from being garbage-collected—either an implicit or explicit reference to the `RecyclerView` instance passed to the component outliving the activity.

We’re not sure what we’re dealing with quite yet, so we start by looking at the `MapImportFragment.kt` class holding the `RecyclerView` mentioned in [Figure 11-29](#). Tracing back to the UI element `recyclerViewMapImport` referenced from the layout file, we bring your attention to something curious:

```
class MapImportFragment: Fragment() {

    private val viewModel: MapImportViewModel by viewModels()

    /* removed for brevity */

    override fun onCreateView(view: View, savedInstanceState: Bundle?) {
        /* removed for brevity */
        recyclerViewMapImport.setOnItemClickListener(
            RecyclerViewItemClickListener(
                this.context,
                recyclerViewMapImport,
                object: RecyclerViewItemClickListener.OnItemClickListener {
                    override fun onItemClick(view: View, position: Int) {
                        binding.fab.activate()
                        single.fab(position)
                    }
                }
            )
        )
    }
}
```

```

    )
}

/* removed for brevity */

private fun FloatingActionButton.activate() {
    /* removed for brevity */
    fab.setOnClickListener {
        itemSelected?.let { item ->
            val inputStream = context.contentResolver.
                openInputStream(item.url)
            inputStream?.let {
                viewModel.unarchiveAsync(it, item)
            }
        }
    }
}
}
}
}

```

❶ In the `MapImportFragment`, we attach a custom click listener to every `ViewHolder` in the `RecyclerView`.

❷ The `Context` then is used to get a `ContentResolver` and create an `InputStream` to feed as an argument for `MapImportViewModel::unarchiveAsync`.

When a user clicks on a particular item in the `RecyclerView`, the Kotlin extension function `FloatingActionButton::activate` is called.

Remember, a common cause for a memory leak is when we accidentally inject an `Activity` or a `Context` into a non-Android component.

If you look closely at the `FloatingActionButton::activate` implementation, you can see that we create an implicit reference to the enclosing class, which is the `MapImportFragment` instance.

How is an implicit reference created? We add a click listener to a button. The listener holds a reference to the parent `Context` (returned by the `getContext()` method of the fragment). To be able to access the `Context` from inside the listener, the Kotlin compiler creates an implicit reference to the enclosing class.

Following the code to the `MapImportViewModel` method, we see the `InputStream` passed down to be able to call another private method in the `ViewModel`:

```

class MapImportViewModel @ViewModelInject constructor(
    private val settings: Settings
) : ViewModel() {
    /* removed for brevity */

    fun unarchiveAsync(inputStream: InputStream, item: ItemData) {
        viewModelScope.launch {
            val rootFolder = settings.getAppDir() ?: return@launch
            val outputFolder = File(rootFolder, "imported")
            /* removed for brevity */
        }
    }
}

```

A `ViewModel` object has a lifecycle of its own and is intended to outlive the lifecycle of the view it is tied to until the `Fragment` is detached. Rather than using an `InputStream` as an argument, it is better to use an application `context`, which is available throughout the life of the application and which can be injected via constructor parameter injection in `MapImportViewModel`.¹ We can then create the `InputStream` right in `MapImportViewModel::unarchiveAsync`:

```

class MapImportViewModel @ViewModelInject constructor(
    private val settings: Settings,
    private val app: Application
): ViewModel() {
    /* removed for brevity */

    fun unarchiveAsync(item: ItemData) {
        viewModelScope.launch {
            val inputStream = app.contentResolve.
                openInputStream(item.uri) ?: return@launch
            val rootFolder = settings.getAppDir() ?: return@launch
            val outputFolder = File(rootFolder, "imported")
            /* removed for brevity */
        }
    }
}

```

Of course, turning on LeakCanary can be disrupting for development if an existing application has many memory leaks. In this case, the temptation might be to turn off LeakCanary to prevent disruption to current work. Should you choose to put LeakCanary on your application, it is best to do it only when you and your team have the capacity to “face the music.”

Summary

There is no doubt that Android benchmarking and profiling tools are powerful. To ensure that your application is getting the most out of analytics, it's best to choose one or two tools as appropriate. It can be easy to get lost in the world of optimizations, but it's important to remember that the largest wins come from making optimizations with the least effort and the largest impact. Likewise, it's important to take current priorities and team workload into consideration.

Approach Android optimizations like a nutritionist, encouraging incremental, habitual changes instead of “crash dieting.” Android profiling is intended to show you what's really happening under the hood, but it's important to remember that the average Android developer must prioritize which issues must be addressed in a world where their time and manpower may be limited.

The hope is that you feel more equipped to handle any potential bugs that may come your way, and that this chapter gives you confidence to start exploring some of these tools on your own applications to see how things are working under the hood:

- Android Profiler is a powerful way to analyze application performance, from networking and CPU to memory and energy analytics. Android Studio caches recorded sessions along with heap dumps and method traces for the lifespan of an Android Studio instance so that you can compare them with other saved sessions.
- Network Profiler can help solve Android problems specific to API debugging. It can provide information useful to both the client device and the server where the data comes from, and can help us ensure optimal data formatting within a network call.
- CPU Profiler can give insight as to where most of the time is being spent executing methods, and is particularly useful for finding bottlenecks in performance. You can record different kinds of CPU traces to be able to drill down into specific threads and call stacks.
- Energy Profiler looks at whether CPU processes, networking calls, or GPS locations in an application could be draining a device's battery.
- Memory Profiler looks at how much memory is allocated in the heap. This can help give insight about areas of code that could use improvements in memory.
- LeakCanary is a popular open source library created by Square. It can be helpful to use LeakCanary to detect memory leaks that are

harder to detect at runtime.

- 1 The `@ViewModelInject` annotation is special to Hilt, which is a dependency injection framework. However, constructor parameter injection can also be achieved with manual DI or with DI frameworks like Dagger and Koin.

[Support](#) | [Sign Out](#)

©2022 O'REILLY MEDIA, INC. [TERMS OF SERVICE](#) | [PRIVACY POLICY](#)