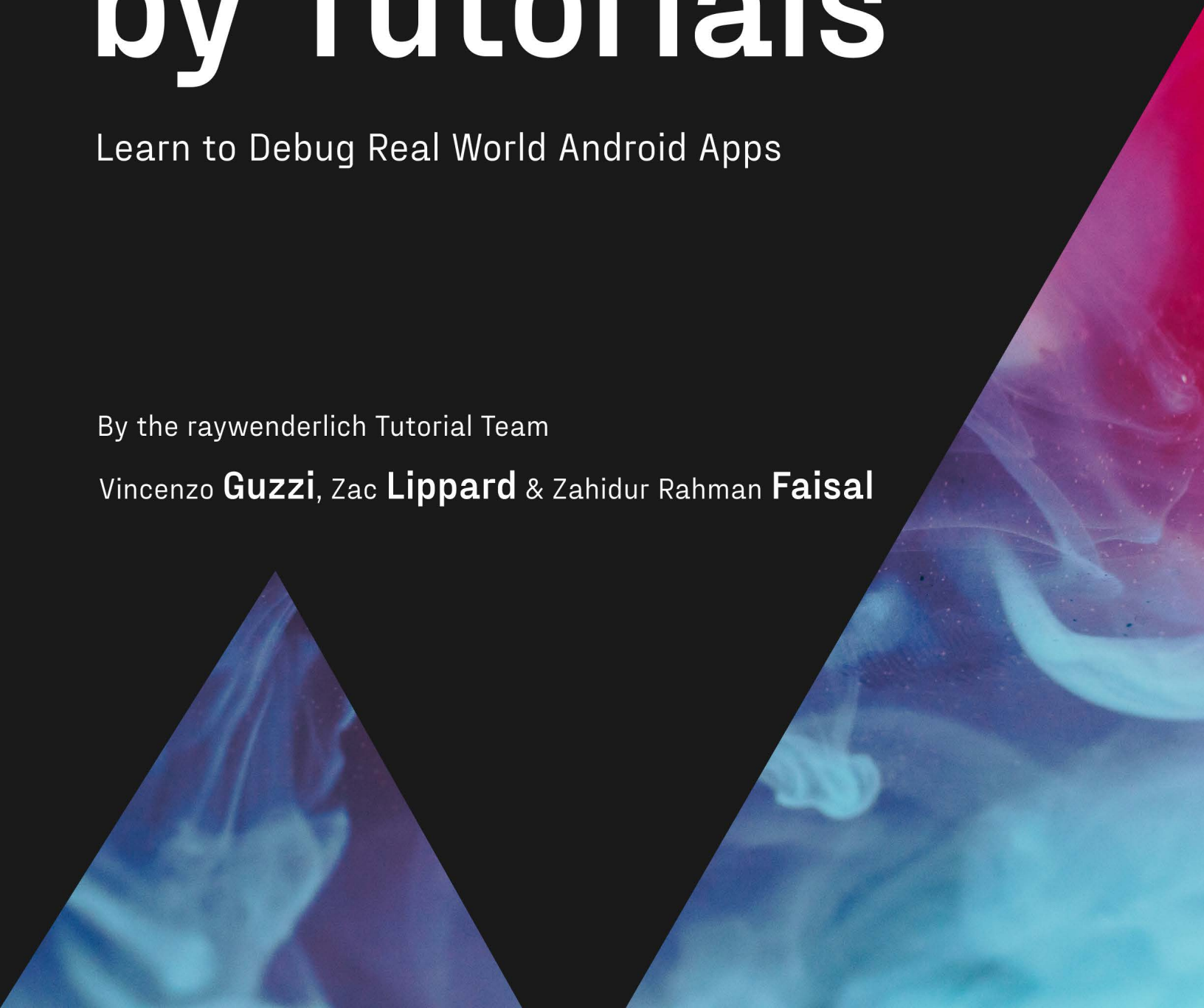
First Edition — Android Studio Chipmunk

Android Debugging by Tutorials

Learn to Debug Real World Android Apps

By the raywenderlich Tutorial Team

Vincenzo **Guzzi**, Zac **Lippard** & Zahidur Rahman **Faisal**



What You Need

To follow along with this book, you'll need the following:

- **Android device**: You'll need a device to test out the book examples. You can either use an emulator (AVD) or a real device.
- **Kotlin 1.6 or later**: This book uses Kotlin 1.6 throughout. The materials may work with older Kotlin versions, but they aren't tested.
- **Android Studio 2021.2.1 or later**: As Android Studio is the main tool for Android app development, you'll need to install AS Chipmunk or a newer version on your machine to try out the book examples. Download the latest version from the Android developer site: https://developer.android.com/studio.

The code covered in this book depends on Android 12, Kotlin 1.6 and Android Studio 2021.2.1 — you may experience issues if you try to work with older versions.

Book Source Code & Forums

Where to download the materials for this book

The materials for this book can be cloned or downloaded from the GitHub book materials repository:

- https://github.com/raywenderlich/adf-materials/tree/editions/1.0

Forums

We've also set up an official forum for the book at https://forums.raywenderlich.com/c/books/android-debugging-by-tutorials. This is a great place to ask questions about the book or to submit any errors you may find.

Dedications

"To my mentors, managers and colleagues who inspire me to sharpen my debugging skills at work, and to my wife Smrity who debugs my everyday life — creating the best version of it!"

— Zahidur Rahman Faisal

"To my wife, Leanna. Thank you for introducing me to the wonderful world of literature, and for proof reading everything I've ever written, including this book."

— Vincenzo Guzzi

"To my wife, Kari. Thank you for always believing in me and helping me realize all that I could do when I set my mind to it. And to my children, Logan and Lana. Thank you for revitalizing that child-like wonder that has kept me passionate about learning and trying new things."

— Zac Lippard

iv About the Team

About the Authors

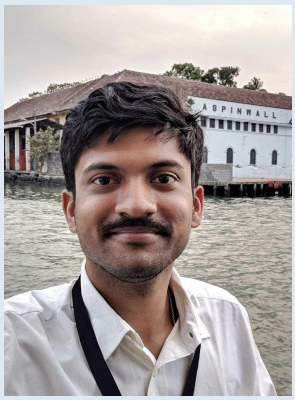
Zahidur Rahman Faisal is the author of this book. He is a digital nomad from Bangladesh in the quest of exploring new technologies as well as new places. He's enthusiastic about mobile engineering and user experience design, he's not a big fan of social media, though! Zahidur loves to spend his free time gaming, reading, wandering around or creating stuff from his own ideas.

Vincenzo Guzzi is the author of this book. He lives in Boston, MA after relocating from Sweden as an Engineering Manager for Spotify. With a background in mobile engineering, Vinny is also passionate about software infrastructure, Flutter, and game development. He loves connecting with new people, get in touch via Twitter https://twitter.com/vguzzi_dev or follow him on Instagram at https://www.instagram.com/vguzzidev/.

Zac Lippard is another author of this book. Zac resides in Murrieta, CA where he works as a Senior Mobile Engineer for Zillow. When he's not writing code and documentation, Zac enjoys hiking with his family, fishing, planet-watching through his telescope, and living vicariously through No Man's Sky. Reach out to Zac on Twitter https://twitter.com/zebdor if you'd like to say "hi" or talk about anything mobile or astronomy related!

About the Editors

Martyn Haigh is the tech editor of this book. Martyn is a seasoned software engineer, currently working at Meta. He started hacking on Android way before it was cool, and has the scars to prove it. When not working, he loves coffee, traveling without a plan, snowboarding, good food and his gorgeous family.



Subhrajyoti Sen is the tech editor of this book. He is a Google Developer Expert for Android and an Android Engineer at Motive where he develops apps to improve the trucking industry. Before this, he has also worked on apps to improve the experience of Indian investors. He believes in the power of Open Source and communities and actively tries to give back. When not writing code, you can find him binge-watching anime, reading up on public policy, or playing Rocket League.

Leanna Guzzi is the editor of this book. She has recently moved to America with her husband and cat. She loves winter, cheese and all things literature. When she isn't empowering young minds and convincing everyone to read, she is planning her next snowboarding or adventure holiday.

Gabriela Angebrandt is the final pass editor of this book. She is a tech enthusiast with a master's degree in Computer Science based in Osijek, Croatia. She got into programming during college and has been developing Android and iOS apps for several years. Her passion for sharing knowledge drives her to contribute to the developer community through books and tutorials. In her spare time, Gabriela attends tech talks and invests time in learning new things from the programming world. When she's not crushing it in the tech world, she enjoys spending time with her fiancé and cat, going into the forest with her horse and playing the piano.

v Introduction

Welcome to the First Edition of *Android Debugging by Tutorials*, created for Kotlin 1.6 and Android Studio 2021.2.1!

Android Debugging focuses on explaining how to debug your app and avoid unexpected app behavior. Once you read this book, you'll be able to find every bug or issue inside your app, fix it and move your app to a higher level. You won't doubt whether your end users will experience weird results or bad app performance anymore. You'll be aware of all issues reproduced at least once and know which steps to take to resolve them.

The most important part is getting familiar with Android Studio's debugging tools. Once you tackle each of them and find out what they offer, you won't ever stop using them!

It's easy to notice and resolve bugs that are visible on the UI or can be experienced while using the app. The problem arises when you have hidden bugs that can happen occasionally or when your app structure or architecture leads to memory or CPU management issues.

Thankfully, Android Studio contains built-in profilers that can present these issues to you in a specific way. You only need to learn how to use and interpret these profilers. Lucky you! This book will teach you everything from scratch.

At the moment of publishing this book, there were no other guiding books based on the Android debugging topic. This is an excellent opportunity for you to learn something new from the debugging field and extend your skills while having everything related to debugging apps in one place.

You'll start with learning debugging basics and simple mechanisms which you can use to find and investigate bugs. After you acknowledge the basics, you'll move to more complex tools. Don't worry, you don't need to install any other software for this book. Everything you need is already prepared for you in Android Studio!

With a lot of practice, this book will make you a debugging expert! But don't stop here! The book contains some great references which you can use to continue your debug journey.

How to read this book

In this book, each chapter teaches about a specific debugging tool. It also contains a simple, practical assignment through which you learn how to use the tools faster and test your existing knowledge.

The book doesn't contain a lot of programming theory. It's more focused on explaining tool mechanisms, their options, actions and results. Understanding these tools is the first step in building more performant apps.

If you want to notice even the smallest detail, read these chapters in order. Chapters are created and sorted so Android beginners can easily follow them. However, if you're an advanced developer, there's a possibility you'd like to skip some chapters. In case that happens, be sure to continue with the starter project attached to the chapter you're skipping to. Every starter project corresponds to the app state as it was at the end of the previous chapter. So, to have everything functional for the next step, remember to switch to the correct app package.

There is only one sample app used in the book called Podplay, but it's full of bugs that you'll try to fix while going through this book. That means don't expect a fully functional and top-quality app at the beginning. This app should provide enough examples to practice inspecting and resolving issues. In case the app looks familiar to you, there is a possibility you have already used it while reading our book *Android Apprentice*.

Currently, the most popular language for building Android apps is Kotlin. It's also Google's preferred language for Android because of its simplicity. Therefore, this book and sample app use Kotlin code to make you up to date with modern technologies. For running the sample app, you need to use Android Studio.

Most of this book's steps focus on manipulating the Android Studio built-in tools and their actions. There are a few places where you'll need to modify the code inside the sample app. You can either type the code in Android Studio immediately, or you can tackle the issue on your own and get back to the book to reveal the solution.

To make the book more interesting and to give you a chance to test what you've learned, we prepared several challenges that you can find near the chapter's end.

Every chapter contains one or more references that you can follow to learn more about related topics.

This book is split into two main sections:

Section I: Debugging Basics

Debugging is the core of every development process. This term represents steps for detecting and fixing issues that can cause unexpected behavior in your app.

In this section, you'll learn:

- **Chapter 1: Getting Started** — In the first chapter, you'll get familiar with the sample app used inside the book. You'll find out how to use the IDE tools and prepare your debugging environment for usage. This chapter will teach you the basics of Android Debug Bridge and how to attach a debugger to a running app.
- **Chapter 2: Navigating Your Code With Breakpoints** — Once you prepare your debugging tools, you'll start with the basics of debugging: breakpoints. You'll learn how to use breakpoints to pause your running app at a certain point and navigate through methods while the app is paused.
- **Chapter 3: Logcat Navigation & Customization** — Through this chapter, you'll find out how the Logcat tool works and to add your logs. Logcat has some great features like customizing stack traces per your needs so you can easily manage logs.
- **Chapter 4: Analyzing the Stack Trace** — Dive deep into using stack traces and tools which help navigate through them. Learn how to read stack traces and find the source of bugs and issues by using links to code lines provided inside the stack trace.
- **Chapter 5: Watches & Evaluating the Code** — After you understand the debugging basics, you'll reveal how to evaluate code by executing code lines while the app runs. This chapter also explains reading and watching variables and how Debug windows work.
- **Chapter 6: Layout Inspector** — If you're wondering how to inspect UI components built inside layouts, Chapter 6 will tell you how. You'll see how Layout Inspector presents the current screen shown in your app. You'll learn how to preview all UI layers and switch between them. Another thing mentioned here is handling accessibility issues with Colorblind tests as an example.
- **Chapter 7: Debugging Databases** — In this chapter, you'll learn about debugging any database included in your app. For this purpose, you'll use the Database Inspector overview. You'll also learn how to execute DAO Queries and custom SQL Queries on a live database.
- **Chapter 8: Debugging WorkManager Jobs With Background Task**

Inspector — The last chapter in this section will teach you how to handle debugging operations running on background threads. You'll use the Background Task Inspector tool, which gives you an option to view and optimize background tasks.

Section II: The Android Profiler

Now that you know how to manage the debugging process, you'll start learning more complex tools. In this section, you'll meet a powerful set of Android Profiler tools. This tool package provides gear for examining app performance in depth.

Along the way, you'll find out about:

- **Chapter 9: Profile CPU Activity** — The opening chapter will introduce you to the CPU Profiler. This huge and very capable tool contains several timelines for tracking CPU activity and records detailed information in the form of system, method and function traces.
- **Chapter 10: Profile Memory Usage** — In this chapter, you'll understand how to use Memory Profiler to inspect memory allocations and investigate where memory leaks are in your app. You'll also focus on capturing heap dumps.
- **Chapter 11: Profile Network Activity** — Go on a journey to create or view an existing API call using Network Profiler. Then, inspect and debug the network call using options provided in the Network Profiler.
- **Chapter 12: Android Energy Profiler** — The last chapter in this book will present how to use Energy Profiler to resolve issues with long-running services in your app. After that, you'll read about wake locks, jobs and alarms. With this knowledge, you'll be able to reduce draining the battery and provide a good user experience for the end user.

1 Getting Started

Written by Zac Lippard

Imagine you finish an important feature for your Android application and you attempt to run it on your device. The app seems to launch, hooray! But, to your dismay the app crashes not a moment later! What do you do now?

Debugging is the core of every development process, and there is no exception with Android. That doesn't mean it has to be a scary feat to undertake.

In this chapter, you'll set up your debugging environment using the built-in tools Android Studio provides. You'll also learn about some of the basics of the **Android Debug Bridge (ADB)** that powers these debugging tools. By the end of the chapter, you'll be able to connect the debugger to the sample app running on either a device or emulator.

Throughout this book, you'll use the PodPlay sample app to learn about the debugging tools available to you.

> **Note**: This book requires knowledge of the fundamentals of Android development. If you're new to Android, please check out our Android Apprentice book for a complete guide to Android development.

Debugging

So, what exactly is debugging anyway? Well, debugging is the process of finding and resolving bugs. A bug can be any defect in the software that causes unexpected errors and crashes.

Fun fact, while the engineering term "bug" was used as early as the 1870s by Thomas Edison, it became a more popular term in the 20th century. In the late 1940s, Admiral Grace Hopper had a team working on the Mark II computer when they noticed a moth trapped in one of the relays. The team had to literally "debug" the system to fix the error!

A **debugger** is a tool that helps software engineers track and monitor a program and change values in memory. Android Studio offers a built-in debugger tool that can help accomplish these tasks. The debugger allows you to add breakpoints to suspend execution of the Android application, monitor memory and CPU usage and provides a whole bunch of other tracking tools. You'll learn

more about breakpoints in Chapter 2, “Navigating Your Code With Breakpoints”.

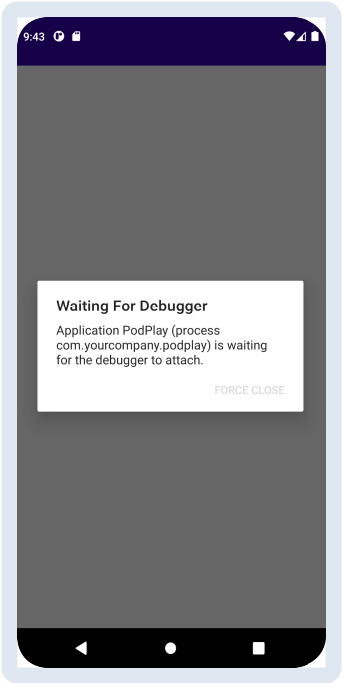
Next, you’ll learn how to connect the Android Studio debugger to an Android application.

Connecting the Debugger

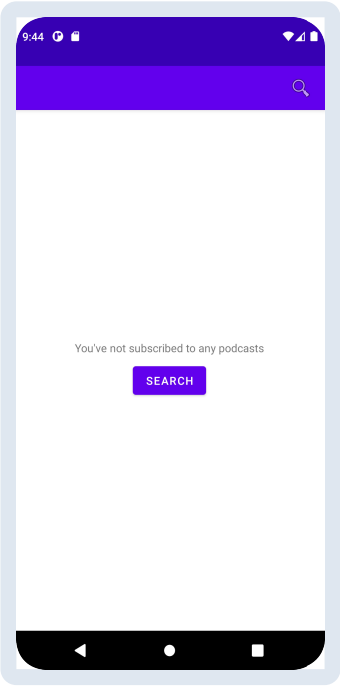
Begin by opening this chapter’s **Podplay** starter project in Android Studio. After the project loads, click the **Debug** icon in the toolbar of Android Studio.

Similar to running the application, Android Studio will build the app if necessary. Then, it will install and run the app on the connected device. The additional step after the app starts is that the debugger is immediately attached to the running application.

If you see a **Waiting For Debugger** alert dialog on the screen, you’ve successfully connected the debugger!



After the debugger is attached, you’ll see the main app screen stating that you haven’t subscribed to any podcasts yet.

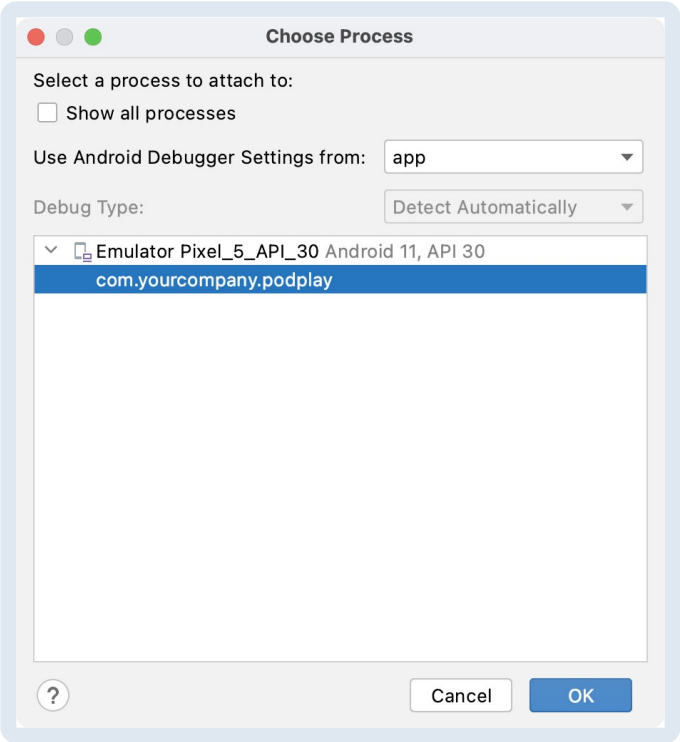


Once the application is running in debug mode, you can stop the application by clicking on the **Stop** icon in the AS toolbar.

Attaching the Debugger

If you already have PodPlay running, you can attach the debugger to the currently running app so you don't have to restart it completely. To do this, simply click the **Attach Debugger to Android Process** icon in the AS toolbar.

A popup will appear, and you can select the running app to connect to.



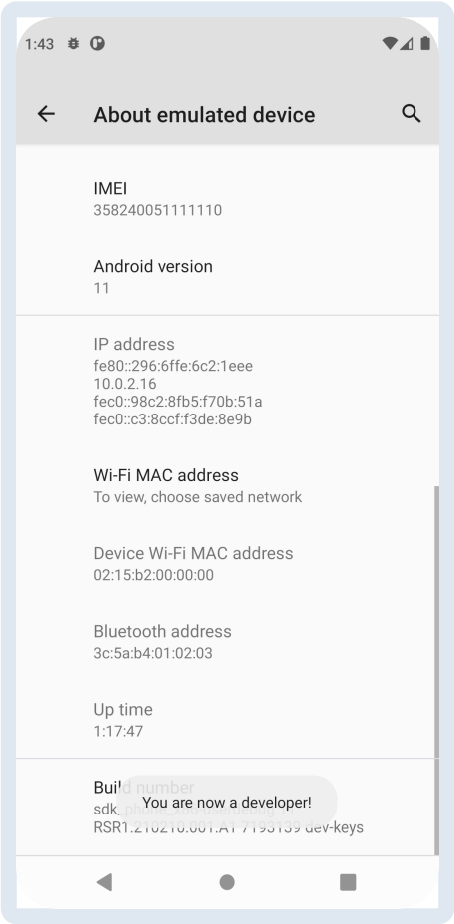
Select the **com.yourcompany.podplay** process and click **OK**. Now you're connected and debugging your previously running app!

Debugging Wirelessly

Android Studio also allows connecting and debugging physical devices over WiFi. The main benefit of **wireless debugging** is the ability to avoid USB connectivity problems, such as driver installation or accidentally disconnecting your device if you have a damaged cable.

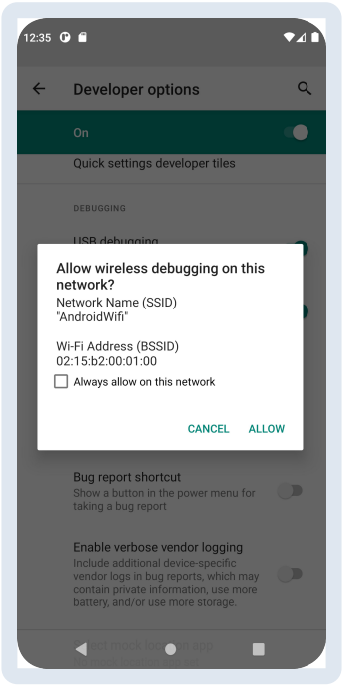
To start, ensure that your workstation and your device are on the same network. The device you're using will need to have **Developer options** enabled.

To enable **Developer options** go to the **Settings** app and choose **About phone**. Scroll to the bottom where the **Build number** option is; if you don't see it, look for **Software information**. Tap the option *seven* times. Congrats! You now have access to the Developer options on the device.

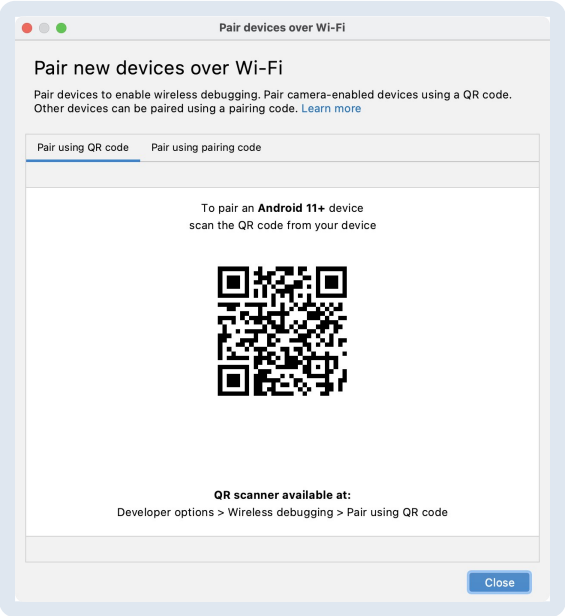


Once you enable the Developer options, go and find that menu, typically under **Settings ▸ Developer options** or **Settings ▸ System ▸ Developer Options**, and look for the **Debugging** section.

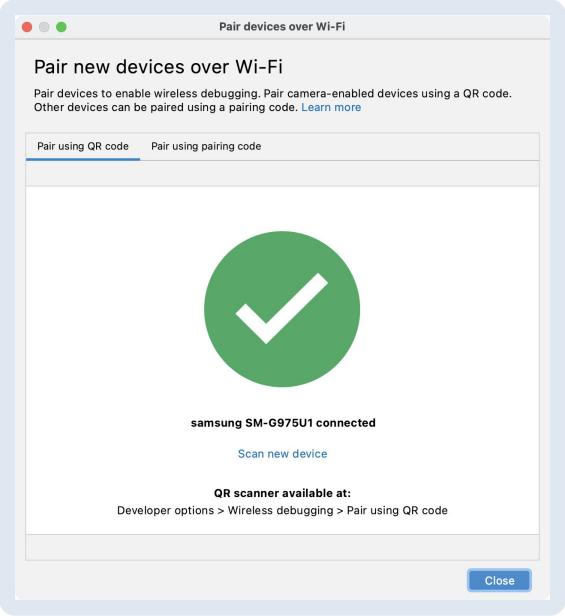
Turn on the **Wireless debugging** option. You'll be prompted with the following dialog to accept wireless debugging on the device's currently connected network:



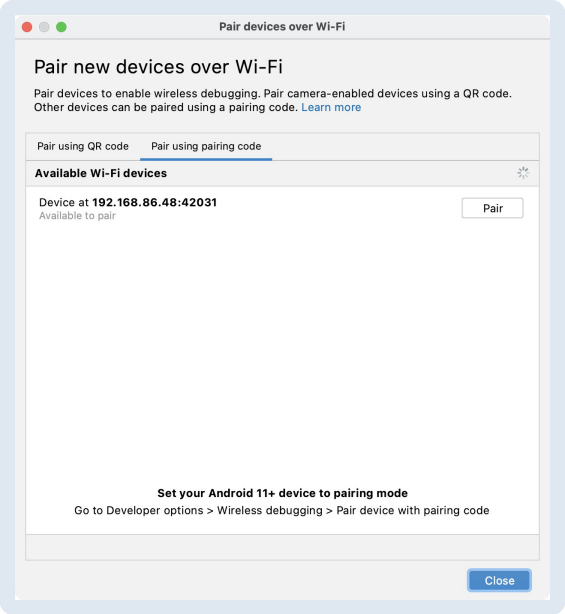
Once your device has enabled wireless debugging, you're ready to connect it to Android Studio! To do so, click the devices dropdown in the toolbar in Android Studio and select the **Pair Devices Using Wi-Fi** option.



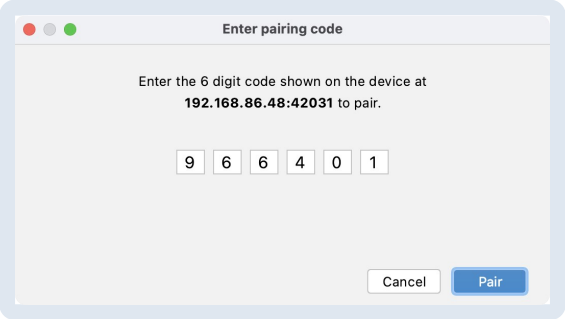
For Android 11+ devices, you can scan the QR code to pair. Open the **Wireless Debugging** option and choose **Pair device with QR code**. Scan the QR code from the Android Studio dialog, and it should automatically pair.



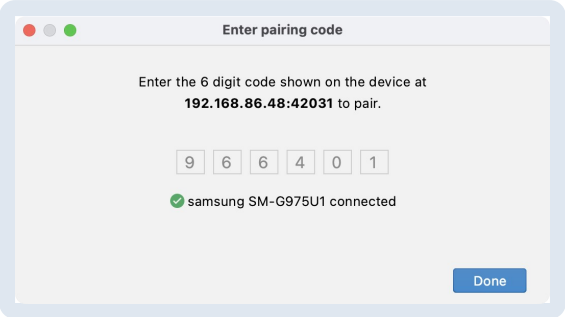
You can pair the device with a pairing code for devices lower than Android 11. To do this, open the **Pair using pairing code** tab in the dialog. On your device, tap **Wireless Debugging** and choose the **Pair device with pairing code** option. A dialog on the device will appear with a code, and the device will appear on the Android Studio pairing devices dialog. Click **Pair** to be able to enter the code.



Enter in the code from your device and select **Pair**.



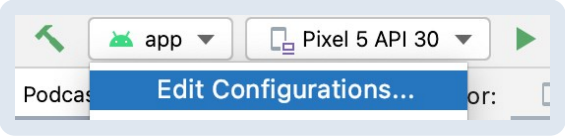
The device should pair, and you'll now be able to run or debug your app wirelessly!



Debugging Multiple Devices

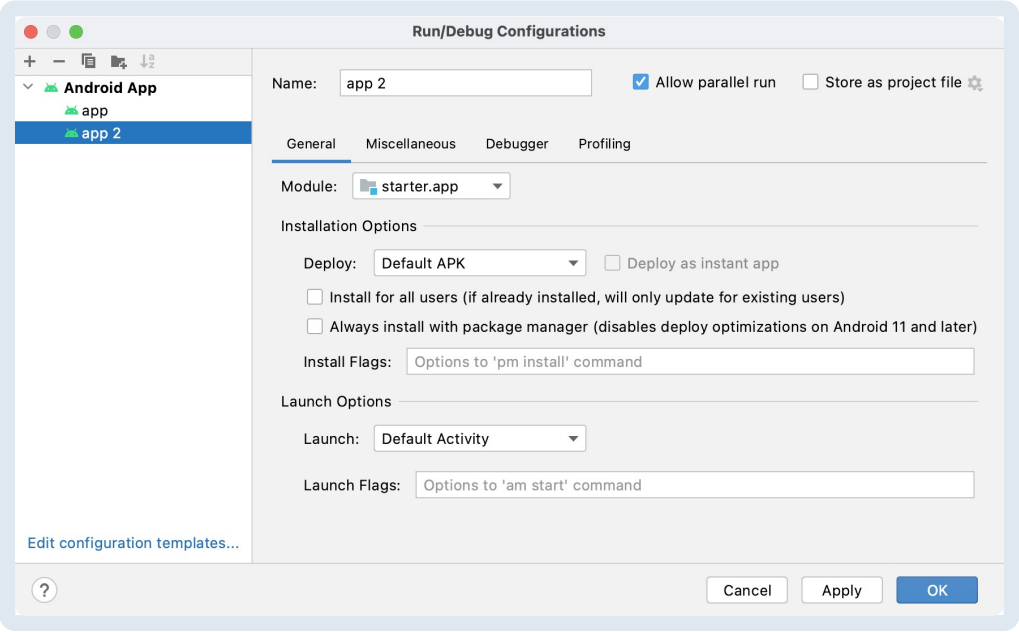
You can also simultaneously connect to multiple devices and debug them in Android Studio. This can be helpful if you need to run a side-by-side comparison of the same app but on different devices.

Firstly, in Android Studio, click the **Select Run/Debug Configuration** dropdown and choose **Edit Configurations...**.



In the **Run/Debug Configurations** window:

1. Select the + icon at the top-right of the dialog to add a new configuration.
2. Choose **Android App** in the left column.
3. Enter in **app 2** for the configuration name.
4. Select **starter.app** for the module.
5. Confirm with **OK**.



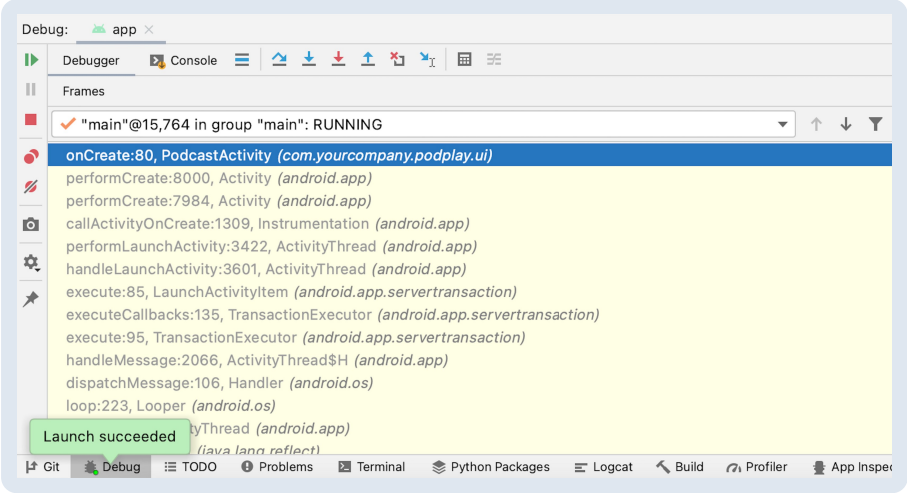
Note: If you plan on debugging more than two devices at once, you'll need to add additional configurations for each additional emulator/device that you've connected.

Open **PodcastActivity.kt** and add a breakpoint inside `onCreate()`:

```
78  override fun onCreate(savedInstanceState: Bundle?) {
79    super.onCreate(savedInstanceState)
80    databinding = ActivityPodcastBinding.inflate(layoutInflater)
81    setContentView(databinding.root)
82    setupToolbar()
83    setupViewModels()
84    updateControls()
85    setupPodcastListView()
86    handleIntent(intent)
87    addBackStackListener()
88    scheduleJobs()
89
90    // TODO: Chapter 11 - Prevent unnecessary network calls
91    performSearch( term: "")
92  }
```

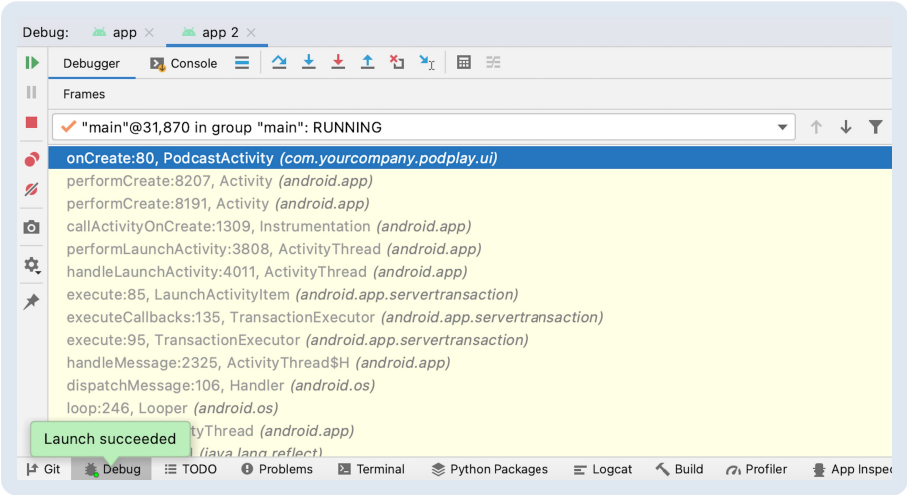
Now, click **app** in the **Run/Debug Configurations** in the AS toolbar dropdown, and then choose the first device you want to debug. Next, press the **Debug** icon to launch the app in debug mode.

When the application first launches, the breakpoint will hit!



Next, while the **app** configuration is still debugging and paused on the breakpoint, switch to the **app 2** configuration in the toolbar dropdown. Select your second device, and then click the **Debug** icon.

Now, the breakpoint will hit again, but notice that this time in the debug window, there is now an **app 2** tab that is active.



This represents the configuration and device pair that you ran the second time. You can toggle between the tabs to continue running the app on each device and set other breakpoints as necessary.

When you're done, press the **Stop** icon in the toolbar and select the option to **Stop All** processes. This will disconnect the debugger from both devices.

Now that you have devices connected for debugging, you're ready to start capturing data from them.

Capturing Device Data

Being able to see what's happening on the device at a given time is a critical debugging technique. Knowing what is happening on the device will give you a better clue as to what may be going wrong in your app in the event of a bug or crash. Some common ways to capture device data are screenshots, recordings and bug reports.

First, you'll learn about screenshots.

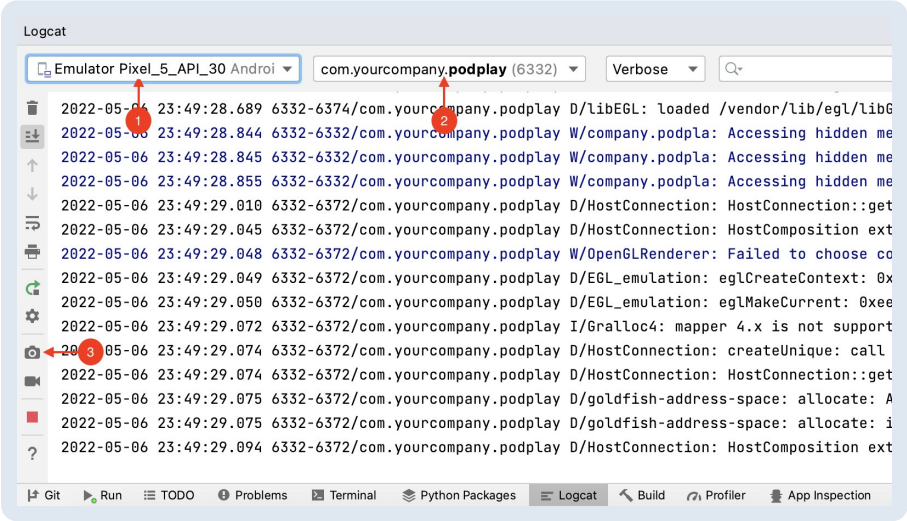
Screenshots

One of the most common capturing tasks you'll need to perform is taking a screenshot of the device. There are a few different ways to take a screenshot of an Android device:

1. From the device itself.
2. In Android Studio.
3. Via the emulator.

You'll need to simultaneously press and hold power and volume-up buttons to take a screenshot from most Android devices. A "snap" will occur when you do this, and a shutter sound will play, indicating that you took a screenshot.

Run the application on the device by pressing the **Play** icon in the AS toolbar. Go to the **Logcat** window by selecting **View ▸ Tool Windows ▸ Logcat**.

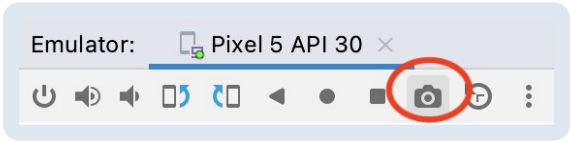


Once the Logcat window appears, do the following:

1. Choose the device in the top-left dropdown.
2. Select the **com.yourcompany.podplay** process.
3. Click the **Screen Capture** camera icon in the left column of the window.

A new dialog window will appear with the screenshot from the device, and you can save the image to your workstation.

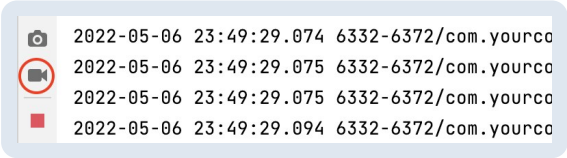
If you're running an emulator, there's another way to grab a screenshot. In the emulator window, tap the **Take a screenshot** icon.



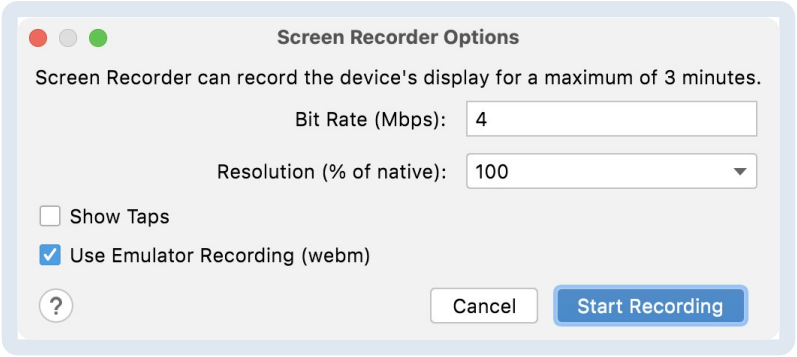
Similar to the Logcat screenshot approach, a dialog will appear where you can save the screenshot to your workstation.

Recording Videos

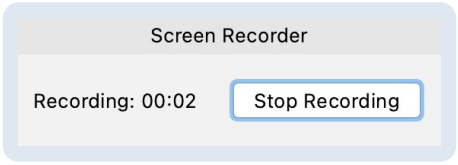
Within the Logcat window, you can also record videos of your device. Press the **Screen Record** video recorder icon in the left column of the Logcat window.



In the **Screen Recorder Options** dialog, you can choose the bit rate and resolution you prefer for the recording or use the defaults provided. The dialog also contains options to show screen taps and set the video format if you're using an emulator.



Click **Start Recording** to start the recording. You'll see a new dialog showing that the recording is in progress.



At this point, you can perform the necessary actions you want to have recorded on your running device or emulator. When you're done, just use the **Stop Recording** button to stop the video. You'll then be prompted to save the recording to your workstation file system.

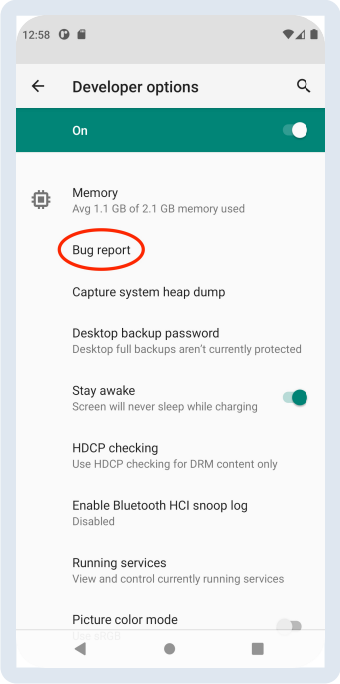
> **Note**: There is a lot more to the Logcat tool than just screenshots and videos! You'll learn more about all the features that Logcat has to provide in Chapter 3, "Logcat Navigation & Customization".

Generating a Bug Report

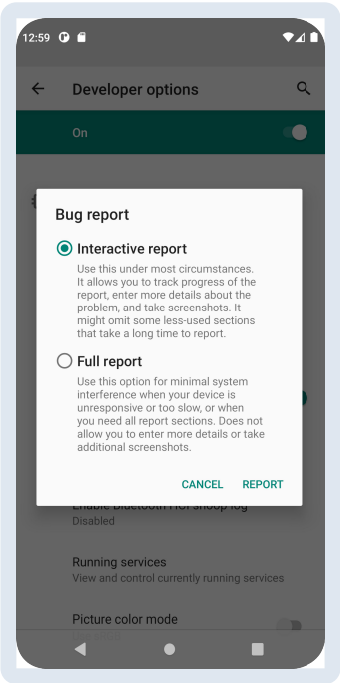
Bug reports are another crucial feature to have in your debugging arsenal. These reports can include things such as stack traces of thrown exceptions or crashes, logs, and other diagnostic information. You can generate bug reports on the device directly or in the emulator menu.

To generate the report on the device itself, you'll need to have enabled **Developer Options**. Go to **Settings ▸ System ▸ Developer Options**, or **Settings**

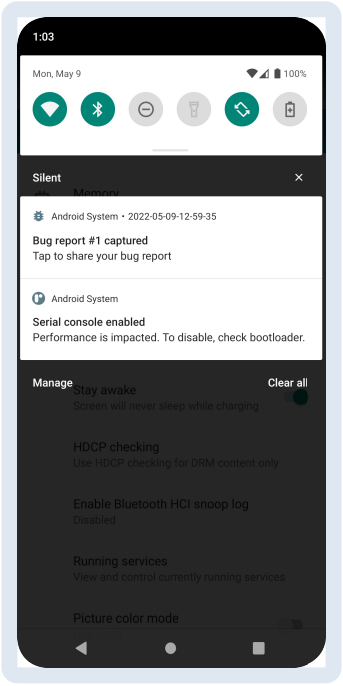
▸ **Developer Options** on some devices, and tap **Bug Report**.



In the **Bug report** dialog, select the type of report you want.

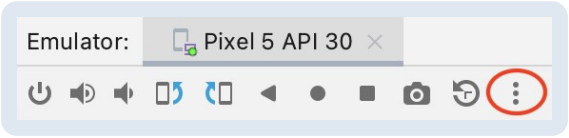


Tap **Report** when you're ready to have the system generate the bug report. While the report is generating, you'll see a notification appear in the status bar. This notification provides information about the progress of the report. When the report completes, you'll see another notification appear with the option to share the bug report.

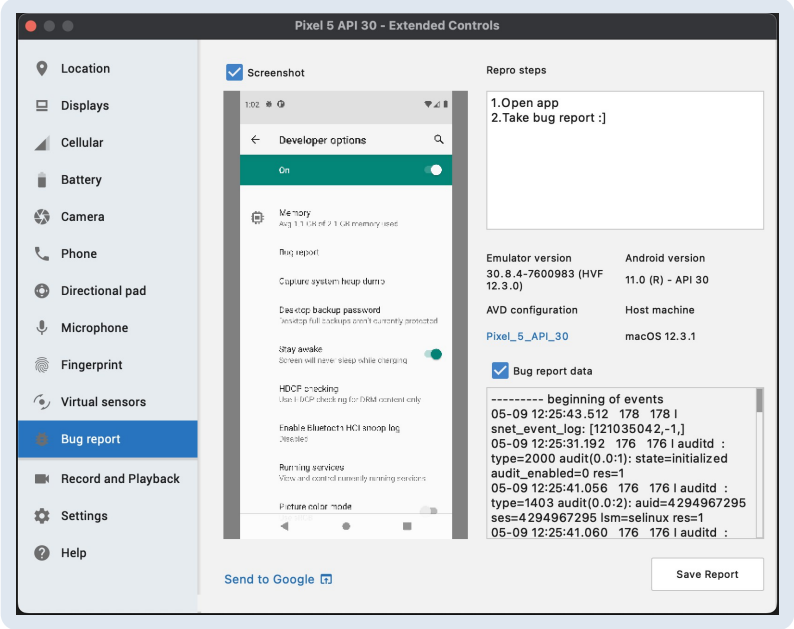


While sharing bug reports from devices is extremely useful, if you don't yet have a bug report but can reproduce a specific error or crash you can generate a bug report directly from an emulator and save it to your workstation.

To generate a report from an emulator, click the **More** option in the emulator window.



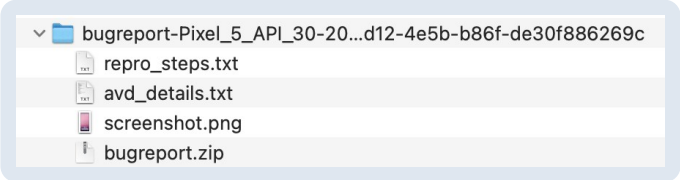
In the **Extended Controls** dialog, click **Bug Report** in the left column. Here, you have the option to add a screenshot, provide steps to reproduce the bug and include the bug report itself.



Select **Save Report**, and you'll be prompted to save the bug report folder on your workstation.

Find the associated `bugreport-DEVICE_ID-DATETIME-HASH` folder and open it. The following files will be in the folder:

- **avd_details.txt**: This file provides detailed information about the emulator, such as name, target type, Android OS, AVD file path, and configuration parameters for the emulator.
- **bugreport.zip**: This zip contains the actual bug report details, including logs, stack traces, and other diagnostic data. The following section — **Reading the Bug Report** will cover this in greater detail.
- **repro_steps.txt**: The steps to reproduce that were entered in the previous **Bug Report** dialog.
- **screenshot.png**: The emulator screenshot when you generated the report.

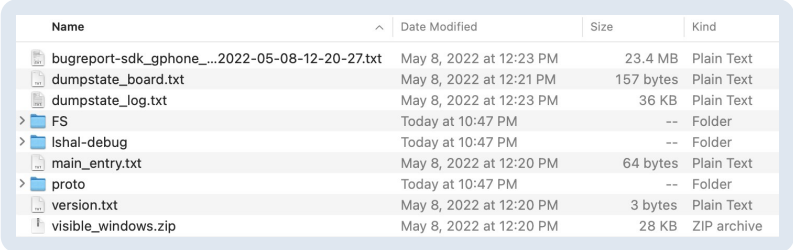


Now that you have a bug report, it's time to inspect it!

Inspecting the Bug Report Folder

Depending on the method of generating and receiving the bug report, you'll typically receive a zip file named either `bugreport.zip` or `bugreport-{BUILD-DATETIME}.zip`. Extract the associated zip file and open up the associated **bugreport** folder. There are several files and folders within, but here are some of the prominent ones:

- **bugreport-{BUILD-DATETIME}.txt**: This is the main file that you'll want to reference when you're attempting to debug your app. It contains error/system logs, system service output and diagnostic information like stack traces.
- **FS/**: This folder contains a copy of the device's filesystem.
- **version.txt**: This is the version associated with the Android release letter.



Name	Date Modified	Size	Kind
bugreport-sdk_gphone_...2022-05-08-12-20-27.txt	May 8, 2022 at 12:23 PM	23.4 MB	Plain Text
dumpstate_board.txt	May 8, 2022 at 12:21 PM	157 bytes	Plain Text
dumpstate_log.txt	May 8, 2022 at 12:23 PM	36 KB	Plain Text
FS	Today at 10:47 PM	--	Folder
lshal-debug	Today at 10:47 PM	--	Folder
main_entry.txt	May 8, 2022 at 12:20 PM	64 bytes	Plain Text
proto	Today at 10:47 PM	--	Folder
version.txt	May 8, 2022 at 12:20 PM	3 bytes	Plain Text
visible_windows.zip	May 8, 2022 at 12:20 PM	28 KB	ZIP archive

Next, you'll take a look at the first file mentioned, the actual bug report.

Reading the Bug Report

As previously mentioned, the bug report text file contains a ton of useful information about what happened in your app and the Android OS during the time the bug report covered. Here you can view things like Android Not Responding (ANR) scenarios, view stack traces for crashes, logs from Logcat and much more!

With ANRs, you can typically associate the logs with the specific ANR stack trace text files typically found in `FS/data/anr/`. These stack trace files can help you better understand what caused the ANR depending on the thread associated with the stack trace file itself. For example:

```
"main" prio=5 tid=1 Waiting
  | group="main" sCount=1 dsCount=0 flags=1 obj=0x7257ad48
self=0xe1006210
  | sysTid=10231 nice=-10 cgrp=top-app sched=0/0 handle=0xef514478
  | state=S schedstat=( 2160085204 978028843 3198 ) utm=89 stm=126
core=3 HZ=100
  | stack=0xff5e8000-0xff5ea000 stackSize=8192KB
  | held mutexes=
  at sun.misc.Unsafe.park(Native method)
  - waiting on an unknown object
  at
java.util.concurrent.locks.LockSupport.park(LockSupport.java:190)
  at java.util.concurrent.FutureTask.awaitDone(FutureTask.java:450)
  at java.util.concurrent.FutureTask.get(FutureTask.java:192)
  at
com.android.settings.dashboard.DashboardFragment.updatePreferenceSt
atesInParallel(DashboardFragment.java:407)
  at
com.android.settings.dashboard.DashboardFragment.updatePreferenceSt
ates(DashboardFragment.java:356)
  at
com.android.settings.dashboard.DashboardFragment.onResume(Dashboard
Fragment.java:212)
  at
com.android.settings.dashboard.RestrictedDashboardFragment.onResume
(RestrictedDashboardFragment.java:138)
```

This stack trace shows that there was an ANR that occurred on the main thread, specifically in `DashboardFragment` of the Settings app.

Android Debug Bridge

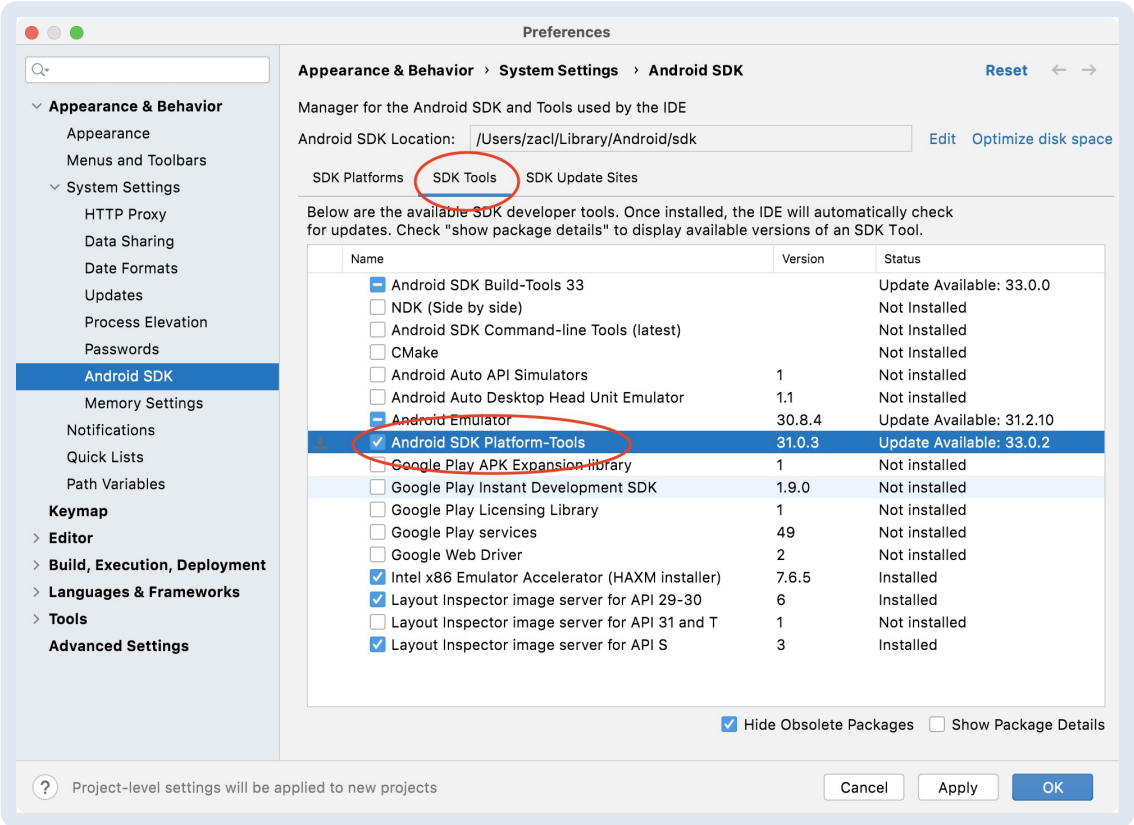
The **Android Debug Bridge (ADB)** is a command-line tool that, at its core, allows you to communicate with your connected device. Most of the debugging tools mentioned in this chapter utilize ADB under the hood. In this paragraph, you'll look at some of ADB's key commands.

Installing ADB

`adb` commands should work out-of-the-box as long as you have Android Studio installed. Android Studio comes bundled with `adb` as part of the **platform-tools** package in the bundled Android SDK. If you receive a "command not found: adb" error when attempting to run `adb`, then you can either download the `platform-tools` package directly or install it via Android Studio.

To download the `platform-tools` in Android Studio click the **SDK Manager** icon in the toolbar:

In the SDK Manager dialog, select the **SDK Tools** tab:



Now check the **Android SDK Platform-Tools** checkbox and click **OK**.

Once you download the package, set your **PATH** environment variable to include `<android_sdk_location>/platform-tools`. Now you're ready to use `adb`!

Starting and Stopping the ADB Server

The first step to using ADB is ensuring the server process is running. The process is typically started when Android Studio is launched. However, it's useful in certain situations to know how to manually start and stop the process.

First, open the Terminal in Android Studio and try out the following commands:

- Start ADB:

```
$ adb start-server
```

- Stop ADB:

```
$ adb kill-server
```

Pretty straightforward, right? :]

If there are any ADB connectivity issues, it is best to try to stop and restart the ADB server process first to see if that fixes things.

Common ADB Commands

While you may not necessarily be using or interacting with ADB directly, it's good to have a handful of commands at your disposal. Many of these commands correlate to the tasks and features that you covered earlier in this chapter:

- Previewing the list of connected devices (either via USB or wirelessly):

```
$ adb devices
List of devices attached
adb-R58MA3DJGKP-nC7KH1._adb-tls-connect._tcp.   device
emulator-5556   device
```

- Launching a specific emulator:

```
$ adb kill-server
$ emulator -avd Pixel_5_API_30 -port 5557
$ adb start-server
```

```
$ adb devices
List of devices attached
emulator-5557 device
```

The code above launches the emulator named **Pixel_5_API_30** on the **5557** port.

- Installing an app:

```
$ adb install MyApp.apk
```

The code above takes the path to the APK as an argument.

- To target commands to a specific device use the `-s` option:

```
$ adb -s DEVICE SOME_COMMAND
...
$ adb -s emulator-5557 install MyApp.apk
```

- Downloading files from device:

```
$ adb pull /file/path/on/device /file/path/on/workstation
```

- Sending files to device:

```
$ adb push /file/path/on/workstation /file/path/on/device
```

- Taking a screenshot, yep, there's even a way to do that in ADB:

```
$ adb shell screencap /sdcard/screen.png
$ adb pull /sdcard/screen.png ~/Downloads/screen.png
```

You can now view the screenshot at **~/Downloads/screen.png** on your workstation.

- Recording a screen capture video:

```
$ adb shell screenrecord /sdcard/recording.mp4
```

Notice the `shell` option used in some of the previous `adb` commands?

You can use `shell` by itself too, and run shell commands directly on the device:

```
$ adb shell
generic_x86:/ $ pwd
/
generic_x86:/ $ cd /sdcard
generic_x86:/sdcard $ screencap /sdcard/screen.png
generic_x86:/sdcard $ exit
$ adb pull /sdcard/screen.png ~/Downloads/screen.png
```

Challenges

Challenge 1: Reading your own bug report

To start reading through the bug report, open a Terminal window pointed at your associated **bugreport** folder.

Here are some great commands for getting information from a bug report:

- To see if there were any associated ANRs occurred:

```
$ grep "am_anr" bugreport-sdk_gphone_x86-RSR1.201013.001-2022-05-
08-12-20-27.txt
05-08 11:30:10.864  1000   510 10422 I am_anr  :
[0,10231,com.android.settings,952745541,Input dispatching timed out
(37c381c com.android.settings/com.android.settings.SubSettings
(server) is not responding. Waited 5001ms for
FocusEvent(hasFocus=false))]
05-08 11:31:57.648  1000   510 10554 I am_anr  :
[0,10231,com.android.settings,952745541,Input dispatching timed out
(ActivityRecord{58a944b u0 com.android.settings/.Settings t13} does
not have a focused window)]
05-08 12:09:19.723  1000   510 13036 I am_anr  :
[0,12519,com.google.android.setupwizard,684211717,executing service
com.google.android.setupwizard/.predeferred.PreDeferredServiceSched
uler]
05-08 12:13:32.705  1000   510 13872 I am_anr  :
[0,12519,com.google.android.setupwizard,684211717,executing service
com.google.android.setupwizard/.predeferred.PreDeferredServiceSched
uler]
```

- To view activities that were in focus to the user and that the user interacted with:

```
$ grep "am_focused_activity" bugreport-sdk_gphone_x86-
RSR1.201013.001-2022-05-08-12-20-27.txt
05-08 11:30:59.125  1000   510 10422 I am_focused_activity  :
[0,com.google.android.GoogleCamera/com.android.camera.CameraActivit
y]
```

- Instances of low memory on the device or emulator can be searched with:

```
$ grep "am_low_memory" bugreport-sdk_gphone_x86-RSR1.201013.001-
2022-05-08-12-20-27.txt
05-06 23:48:00.192  1000   510   587 I am_low_memory: 4
```

To make these commands work, you need to modify them a bit by changing the bug-report file name. Good luck!

> **Note**: **grep** is a command-line utility used to search for patterns in plain-text files. You can learn more about grep on the GNU Manual.

You can do a lot more with just the bug report text file. The commands mentioned just scratch the surface on unlocking your full debug potential.

Challenge 2: Throwing an exception

Now that you have some of the basics of debugging an Android application under your belt, it's time to try it out!

Try throwing an exception in the Podplay starter or final project and see if you can generate a bug report and find the stack trace within it.

Key Points

- Run an app in debug mode to investigate issues in your app.
- Attach the debugger to an already running app without restarting it.
- Android Studio allows debugging the app by wireless connection to your device.
- There is an option to debug multiple devices simultaneously.
- Capture device data via screenshots, video recordings and bug reports.
- Android Debug Bridge is command-line tool to run low-level debug commands.

Where to Go From Here?

Congrats on starting your debug journey. There's more to come in the following chapters, and you'll dive deeper into the different aspects of debugging Android apps. As you work through the chapters in this book, you'll become well equipped to handle almost any error or crash that comes your way.

ADB is a powerful tool to use, and this chapter only demonstrates the tip of the iceberg. Read the Android Developers ADB user guide to learn more about all the commands provided by ADB.

2 Navigating Your Code With Breakpoints

Written by Vincenzo Guzzi

Breakpoints are used for debugging purposes in software development. A **breakpoint** is an intentional place or pause in a program where you can suspend your program or data can be retrieved and logged.

Breakpoints are the backbone of any good IDE and a powerful tool in a developer's toolbox; Android Studio is no different!

If you've worked with any Android code before, you're sure to have used these useful little red circles many times to help locate a bug or simply to try and get a better understanding of your code.

You might think that you know all there is to know about breakpoints in Android Studio, but just breaking on a code line only scratches the surface of how powerful breakpoints can be.

This chapter will teach you how to become a breakpoint power user.

You'll learn:

- How to set breakpoints and navigate your paused app.
- What conditional breakpoints are and how to use them.
- How to utilize unsuspended breakpoint conditions to help locate bugs.
- How to use breakpoints to log helpful information.

Setting Your First Breakpoint

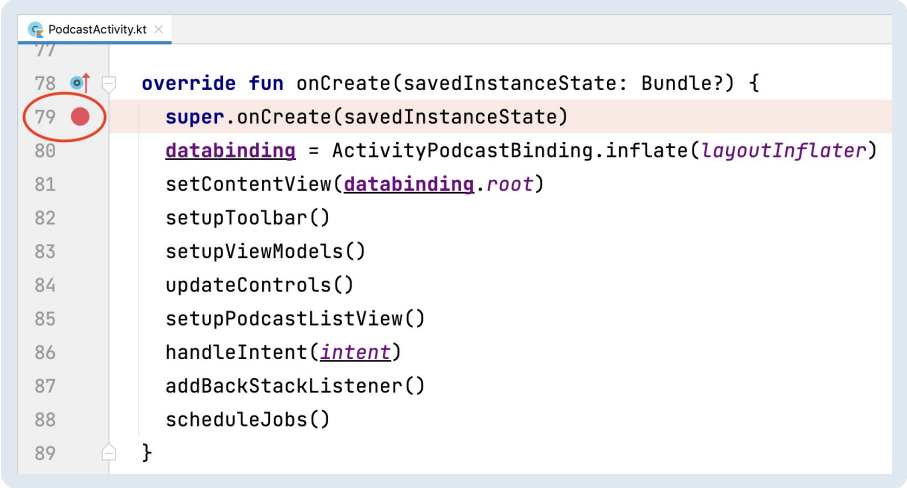
There are several different types of breakpoints you can use in Android Studio. The most common type of breakpoint, and the one that you're probably the most familiar with, is a breakpoint that stops the execution of your app code at a specific place. While your app is in a paused state, you can view variable values, evaluate code expressions and continue executing your app line by line.

Open the **Podplay** starter project in Android Studio and navigate to **PodcastActivity.kt**.

Add a breakpoint on the first line within `onCreate()` by clicking inside the left column next to the line of code that states

`super.onCreate(savedInstanceState)`.

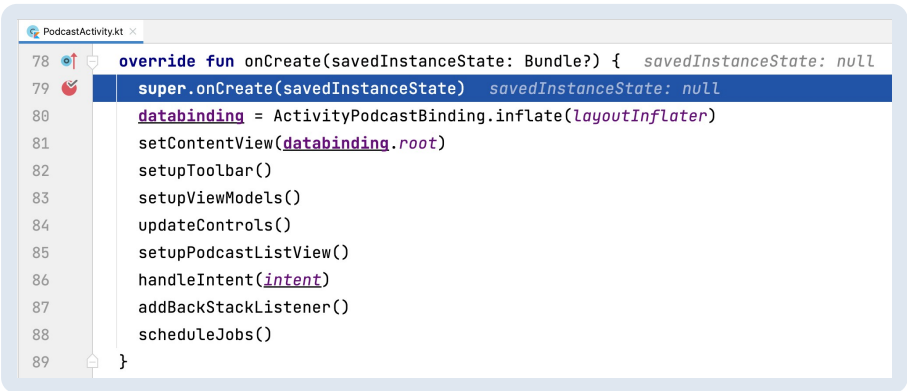
A red circle will appear on that line in the column, like this:



You've now created a **suspended breakpoint** on that line of code. Your app's execution will pause when it gets to that line inside `onCreate()`.

Run your app in **debug mode** by going to the Android Studio toolbar and clicking the **Debug** button:

Your app will load and pause on the breakpoint that you just set. Android Studio will highlight the line of code that it's currently paused at:



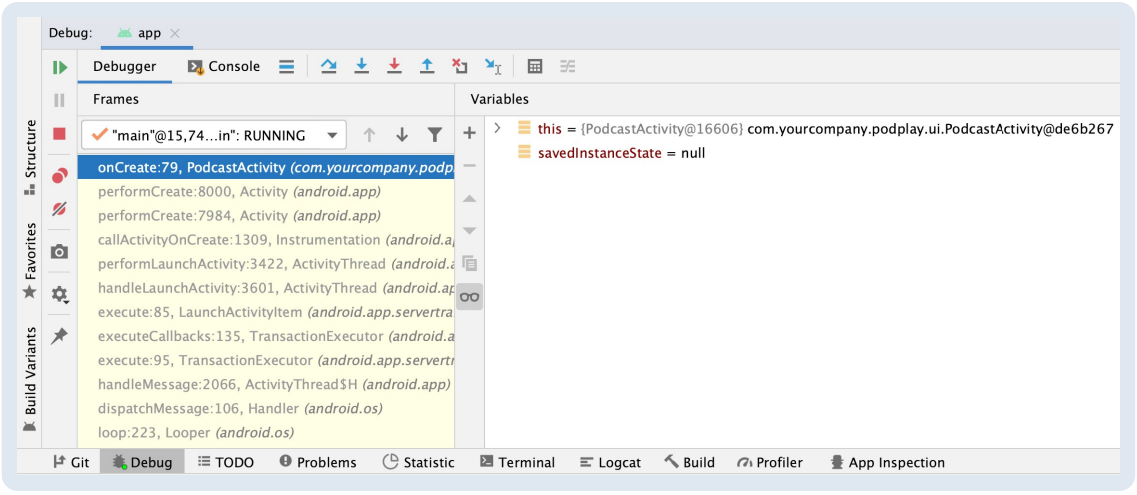
This indicates that the code has paused *before* executing this specific line. You can't interact with your app as it's currently frozen in a suspended state.

Becoming Acquainted With the Debug Window

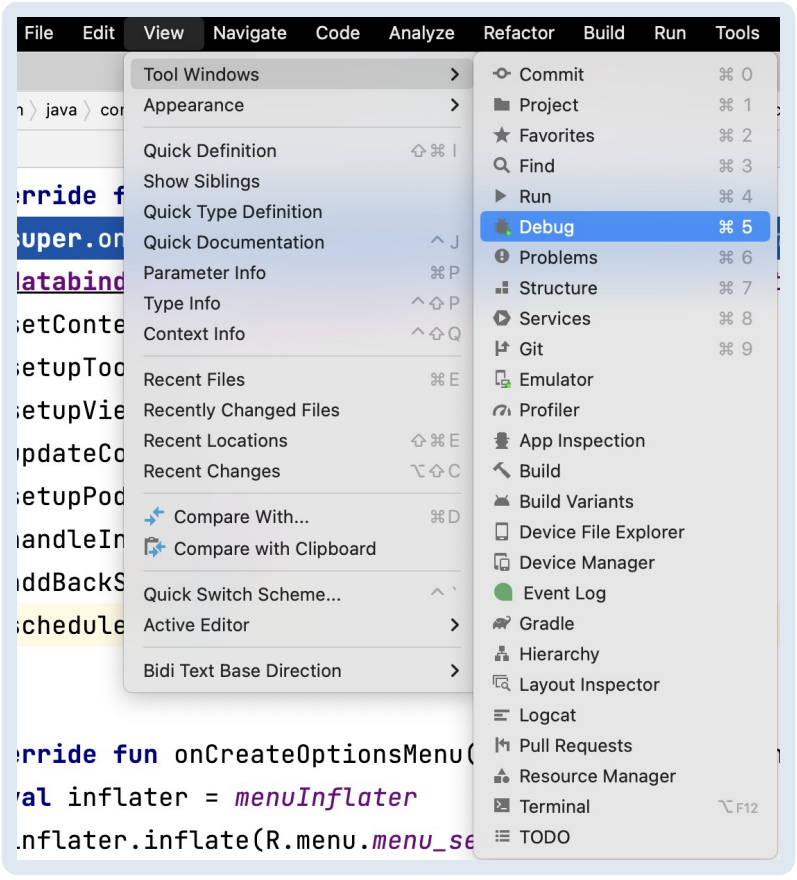
When your code runs in debug mode, you have access to the **Debug Window**.

This is your command center for debugging; it's where you can observe your variables, threads and logs. It's also where you can execute debugging commands such as stepping through your code's execution line by line after it's suspended on a breakpoint.

The Debug Window will automatically appear at the bottom of your Android Studio window when a breakpoint is hit, and it'll look like this by default:



If you don't see this window, toggle it by going to **View ▸ Tools Windows ▸ Debug**.



Stepping Through Your Code

You can find code execution buttons at the top of the Debug Window:

Here is what these buttons do:

1. **Show Execution Point**: Clicking this button will focus Android Studio on your current execution point. If you navigate away from your execution point while your program is paused to investigate another class or method, this button will re-orientate you back to your execution line.
2. **Step Over**: This executes the currently highlighted line of code and moves on to the following line within the current method scope.
3. **Step Into**: By clicking this button the debugger steps into the method on the execution line and starts executing its logic along with switching the scope. The **Frames** pane will open a new stack frame, and your execution line will move into that method. You'll learn about this in the next section.
4. **Force Step Into**: When running *Step Into*, Android Studio will usually only do so when the function that you're stepping into is your own code. *Force Step Into* will tell Android Studio to step into the method no matter what, even if the source code can't be fully indexed. You can learn more about indexing at https://www.jetbrains.com/help/idea/indexing.html.
5. **Step Out**: This will do the opposite of Step Into. The debugger steps out of the current scope, moving one level higher.
6. **Drop Frame**: Similar to Step Out, although you'll return to the code line before the frame began instead of after the frame has completed execution, effectively time traveling back in your program.
7. **Run to Cursor**: This will resume executing the program until it reaches the line of code wherever your cursor is placed (as if there was a breakpoint on that line).

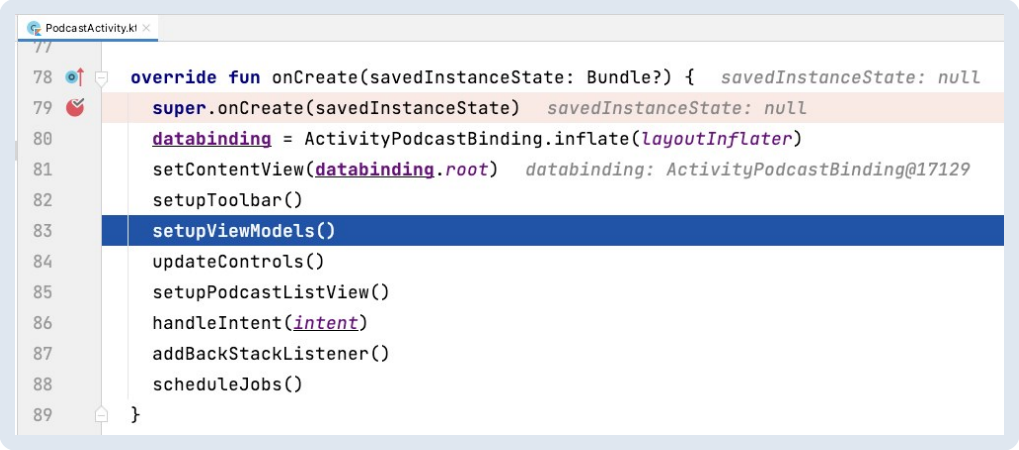
On the left of the Debug Window, you have:

- **Play**: Resumes executing your program until it reaches the next breakpoint.
- **Stop**: Stops debugging your app.

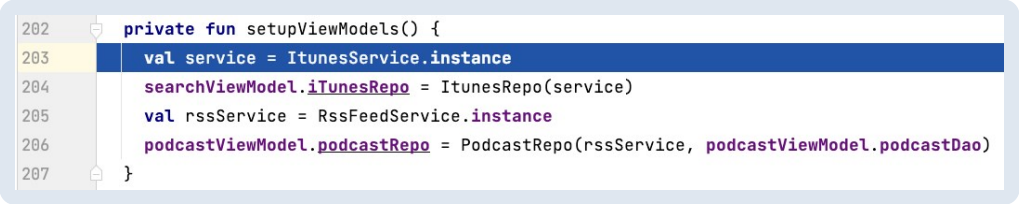
Next, to put these buttons into practice, you'll navigate your code using them.

In the previous step, you entered debug mode with your code execution paused on the `super.onCreate(savedInstanceState)` line inside `onCreate()`; if you're not in debug mode, go to the AS toolbar, click the **Debug** button again and get back to that breakpoint.

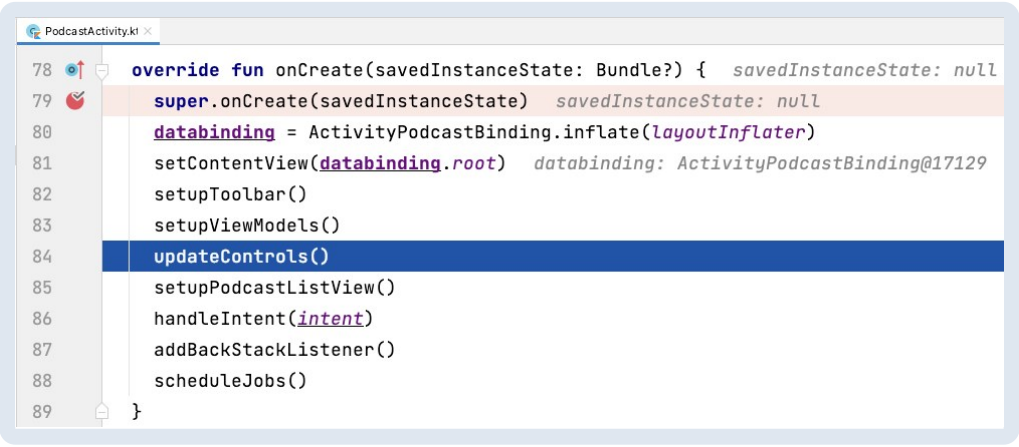
Click **Step Over** four times. Your code execution line moves down within the `onCreate` scope and ends up on line 5 of the function: `setupViewModels()`.



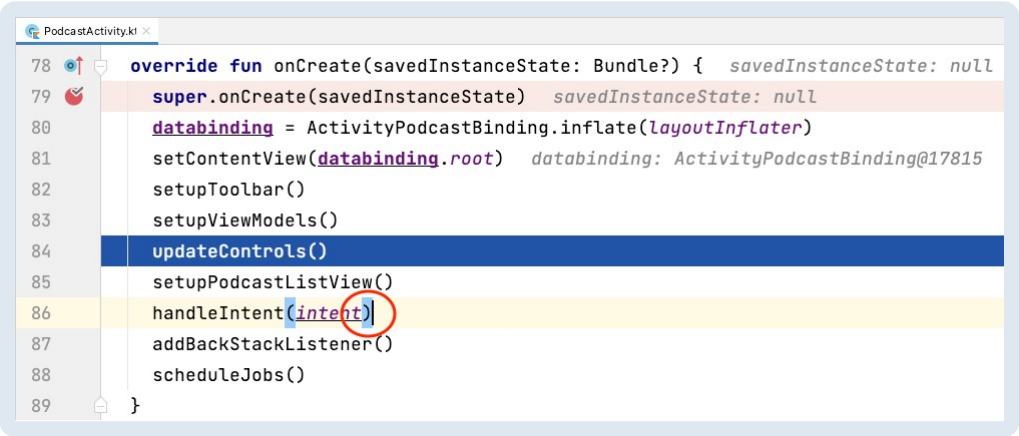
Now, step into the highlighted method by clicking **Step Into**. Your code execution moves to the first execution line of `setupViewModels()`, like so:



Use the **Step Out** button and you'll end up back inside the `onCreate` scope *after* `setupViewModels()` has executed.



Now, go to the line of code that calls `handleIntent(intent)` inside `onCreate()` and click so that your cursor is at the end of that line:



Click **Run to Cursor**, and your execution will resume until it reaches that line. This is a handy button to use if you want to quickly navigate to a line of code without setting a breakpoint there.

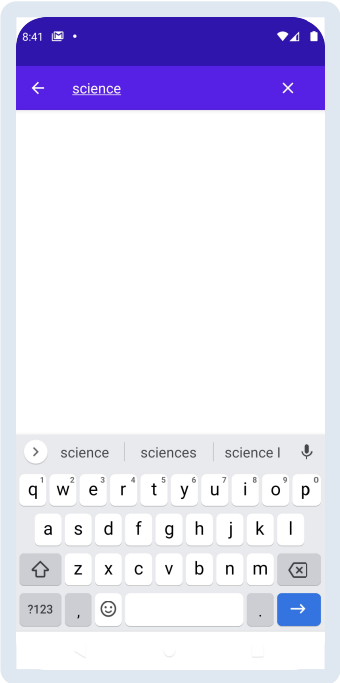
Great job! You've learned the basics of code navigation in debug mode. Click **Stop** to stop your debug session.

Next, you'll learn how about stack frames and how to drop them.

Navigating Stack Frames While Debugging

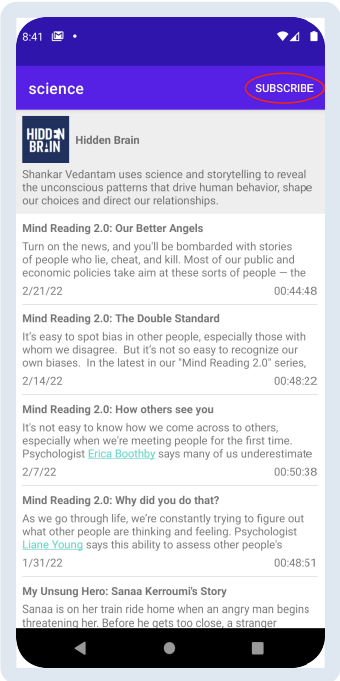
Remove your breakpoint from `onCreate()` inside `PodcastActivity` by clicking the breakpoint circle; you no longer want your code to pause execution on that line.

Relaunch the app by clicking **Play** and subscribe to two podcasts so that they appear on the home screen of your app. Do this by tapping the search icon in the app bar, entering a phrase such as "science" and pressing the **Return** key on your keyboard to start searching for results:

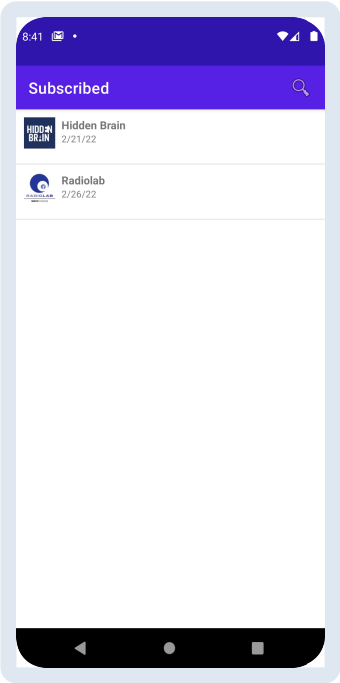


Now, tap a podcast within the list of results.

Then, tap **SUBSCRIBE** at the top right corner of the screen:

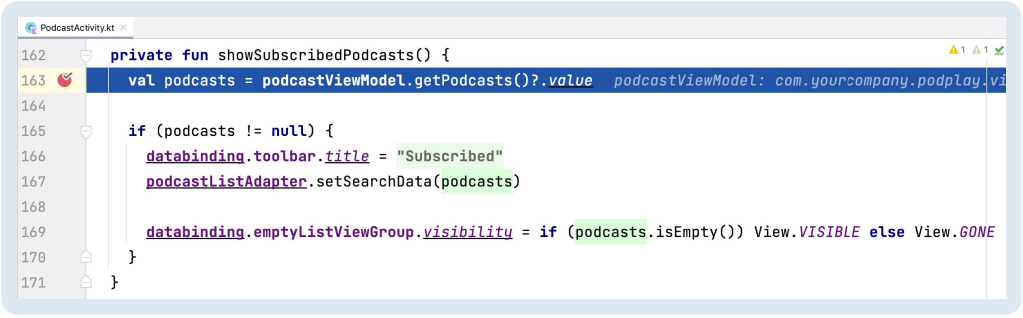


Do this two or more times until you have a selection of subscribed podcasts in your home screen view:

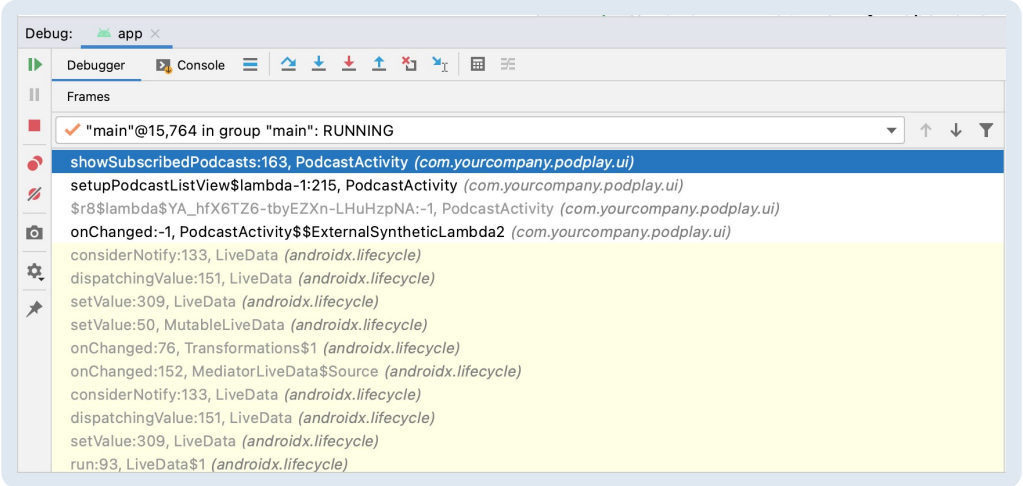


Now, back in Android Studio, further down, inside `PodcastActivity` find `showSubscribedPodcasts()`; place a breakpoint on the first line inside this method's scope where the code is calling `getPodcasts()` from `podcastViewModel`.

Run your app in debug mode and your code execution will pause on that line:

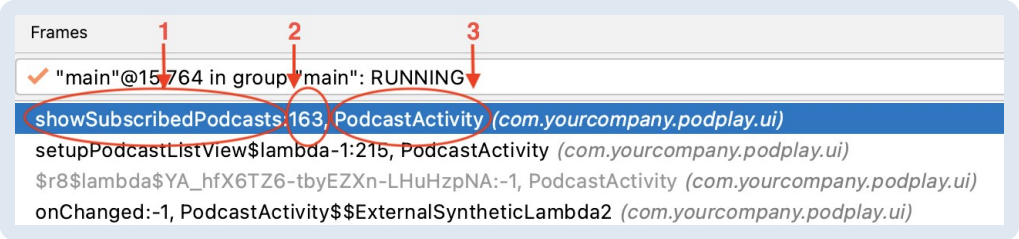


Now, bring your attention to the **Frames** pane inside the **Debug Window**:



Each frame is a list item that represents the scope of a method. The frame you're viewing is highlighted in blue; right now, it's the frame of `showSubscribedPodcasts()`; the scope where your code execution has paused at your breakpoint.

Frame list items show useful information that dictates where it's located:

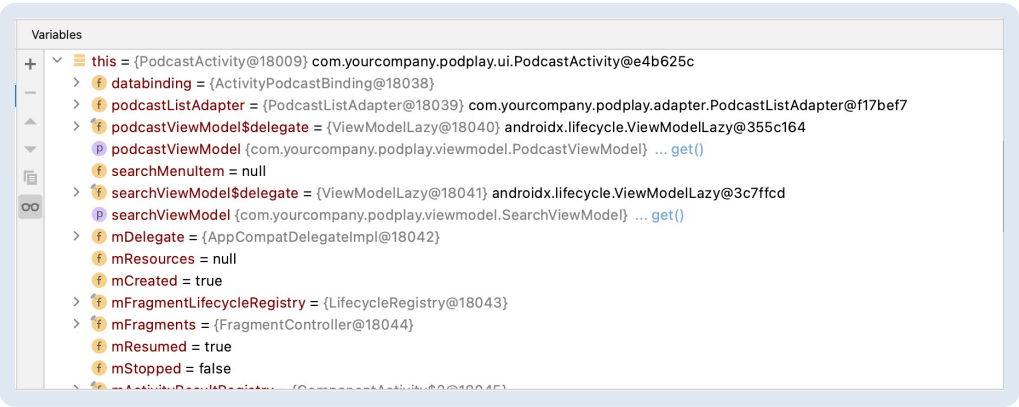


Here is a breakdown:

1. **showSubscribedPodcasts**: The scope in which the frame is located.
2. **163**: The code line number where the code execution is paused within that frame.
3. **PodcastActivity**: The code's class.

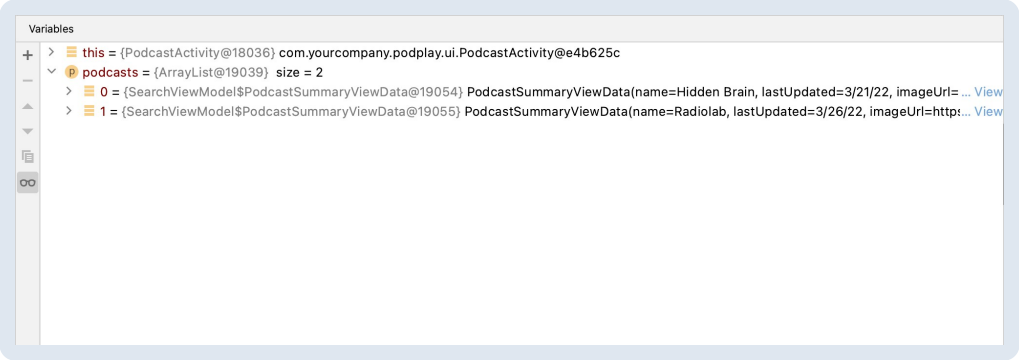
On the right of the Frames pane, you'll see the **Variables** pane. This pane shows the visible variables available to the current frame.

Click the chevron next to **this** in the Variables pane to see a list of all of the variables and values available to the frame's context.



Click **Step Over** to execute your currently suspended line of code.

Your code moves down the scope, executing `getPodcasts()` from the view model. Now your Variables pane shows a `podcasts` variable which is only visible to the active scope of `showSubscribedPodcasts()`. Expanding the list using the chevron again will show you the podcasts that you subscribed to earlier:



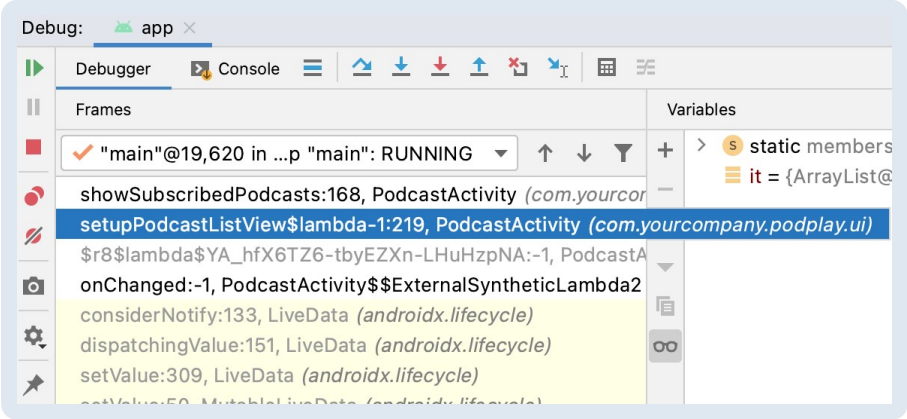
Imagine for a second that you accidentally clicked **Step Over** and you wanted to debug `getPodcasts()`; by stepping over, you've executed that method without going into the scope via **Step Into**.

There is still a way to step into `getPodcasts()` without restarting the app by using the **Drop Frame** feature.

Using Drop Frame to Travel Back in Time

Drop Frame does precisely what the name dictates. It drops the current frame and reverts the code execution to the previous one.

Look at the **Frames** pane and notice that the previous frame in the stack is a frame with the `setupPodcastListView`'s scope. Click that frame and focus on it to expand the frame box:



Notice it holds information about where the first call of `showSubscribedPodcasts()` was.

As you learned earlier in this chapter, the **Drop Frame** action stands next to the **Step Out** action. Now, use **Drop Frame** and click **Step Over** once.

Magic!

You've traveled back in time in your program's execution to the first line of `showSubscribedPodcasts()`.

The **Drop Frame** button can save a lot of time when debugging as, in most use cases, it means that you don't have to restart your app.

The way it works is by caching each of the frames so that they can be reloaded when required. When using this feature, bear in mind that if you were in the middle of a long function that had done a lot of intermediate work, for example, modifying the state of the current class, that wouldn't get undone when you drop the frame; so use **Drop Frame** with that caveat in mind!

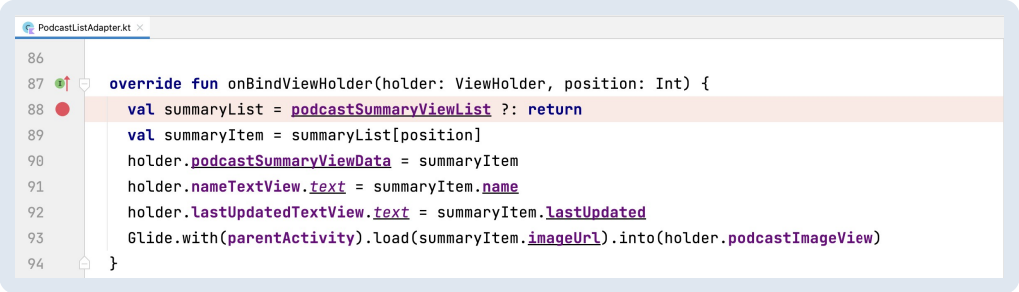
Conditional Breakpoints

Until this point, you've learned how to mark a code line with a default suspend breakpoint that simply pauses your code execution when it's hit.

Now, you'll look at some more advanced features of breakpoints; features that let you control when the breakpoint is hit and what it should do when that happens.

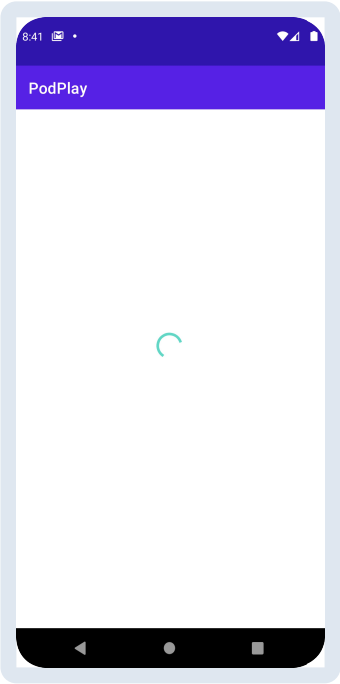
You'll need some more tricky code to debug for this section. Remove your existing breakpoints and stop your code from executing by clicking **Stop**.

Open **PodcastListAdapter.kt** and scroll down until you get to `onBindViewHolder()`. Place a breakpoint on the first line of code within that function:



Right now, you want to debug the logic of `PodcastListAdapter` when the user searches for podcasts.

Run your app in debug mode and wait for your program to reach the breakpoint.



Something has gone wrong!

The code shouldn't have paused here; you haven't entered a search term yet. You might be wondering what is going on.

What happened is that the `PodcastListAdapter` is being used in two places; the first is for displaying the user's list of subscribed podcasts, and the second is for showing the podcast results of a search term.

This is quite a common use case in development; reusing code is actually encouraged. So, how do you ensure that you're debugging the correct adapter? With **conditional breakpoints**, of course!

Dependent Breakpoints

To solve the problem of code reuse, you can use a condition called **breakpoint dependency**. This is where we can set up a breakpoint only to be enabled if a previous, different breakpoint has been hit.

Go back to **PodcastActivity.kt**, scroll down until you find `performSearch()`.

> **Note**: if you aren't a fan of scrolling through large classes like this one, you can press **Shift** twice to search for terms in your project. Just type the method name in the search box to save yourself some time.

This method updates the `PodcastListAdapter` with podcast search results with

this line of code:

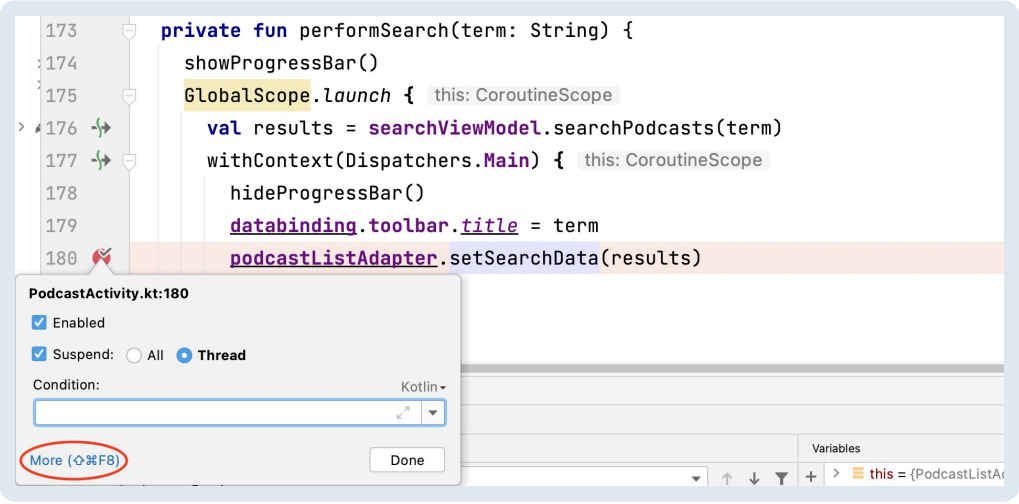
```
podcastListAdapter.setSearchData(results)
```

This is a perfect example of a notifier for suspending on `PodcastListAdapter`. If the system executes it, you know that the user has input a search term and the adapter is about to be repopulated.

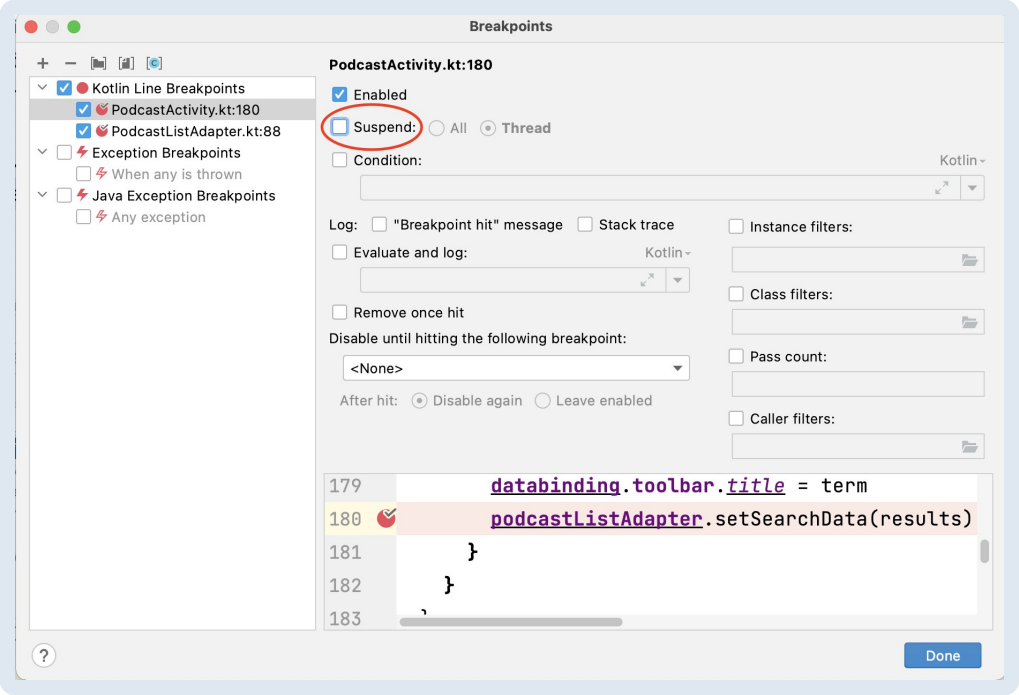
Place a breakpoint here. You don't want the code execution to actually pause on this line of code, so here is where you'll use a new type of breakpoint: An **unsuspended breakpoint**.

An unsuspended breakpoint won't pause your code execution when hit; this can be used for many practical use cases, one of them being logging, which you'll get to later. For now, you'll simply use this as a trigger for enabling your other dependent breakpoint.

To use this type of breakpoint, right-click it and select **More** at the bottom of the breakpoint window:



This will open up a lot more settings for your breakpoint. For now, uncheck the box that says **Suspend**:



Click **Done**. Your breakpoint is now an unsuspended breakpoint; it's turned from red to orange to indicate this.

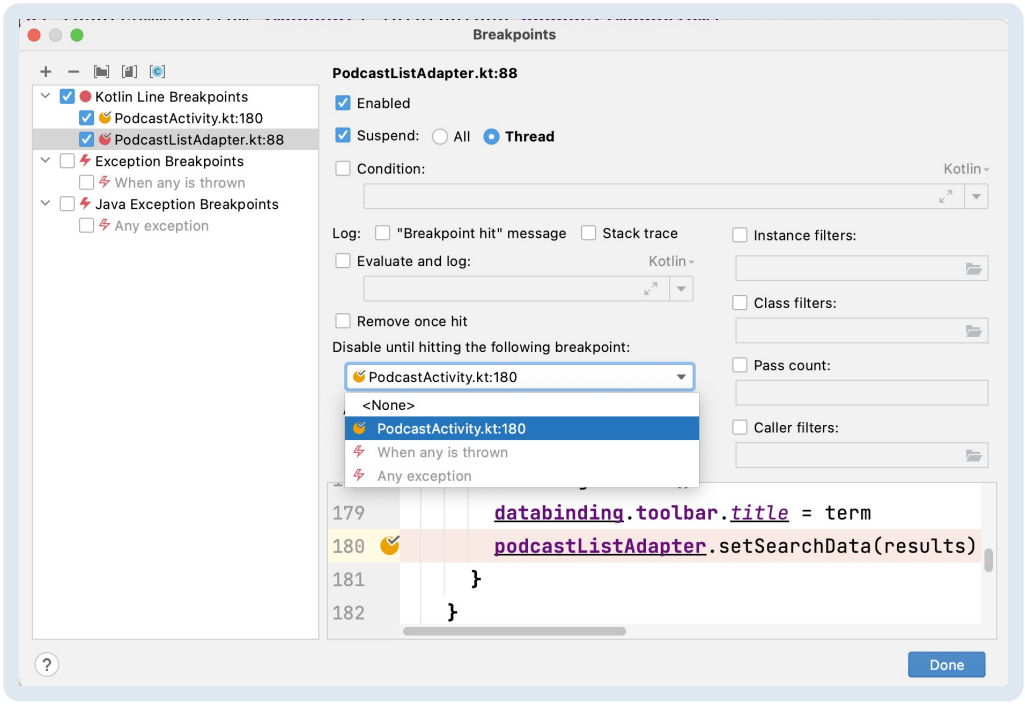
> **Note**: A nice shortcut for placing an unsuspended breakpoint is to press **Shift-click** instead of just clicking next to a line of code.

Next, open **PodcastListAdapter.kt** again and go to your breakpoint in `onBindViewHolder()`.

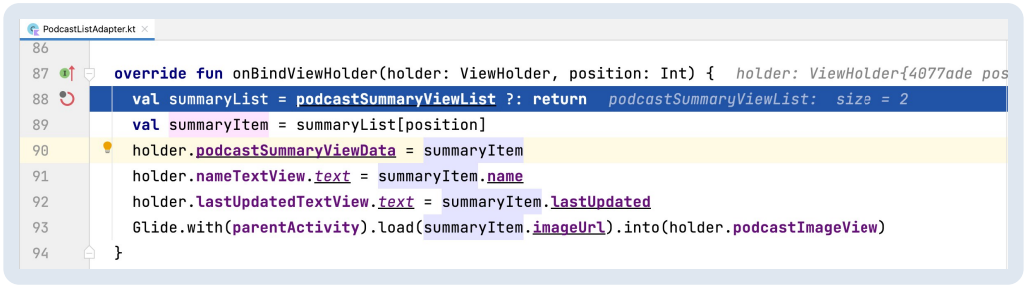
Open the settings of this breakpoint by right-clicking it and clicking **More**.

In the dropdown underneath "Disable until hitting the following breakpoint:", select your `PodcastActivity` unsuspended breakpoint.

Click **Done** to close the Breakpoints window.



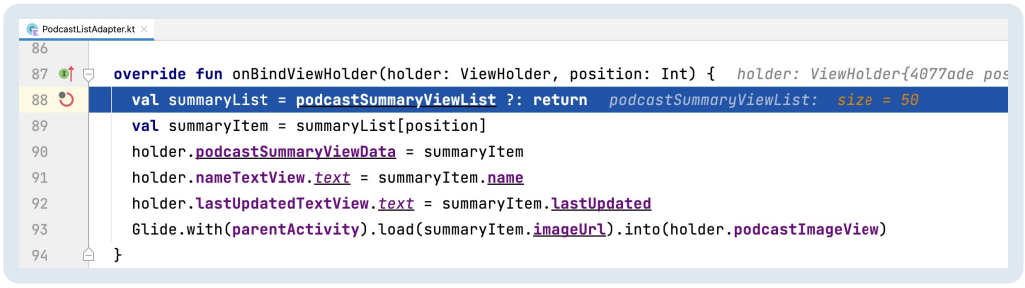
Your `onBindViewHolder` breakpoint will now have a red border:



This indicates that it's currently not enabled. It'll become enabled and, in turn, switch to solid red when the debugger reaches the dependent breakpoint in `PodcastActivity`.

Rerun your project in debug mode. Now, when you launch the app, your code won't stop executing.

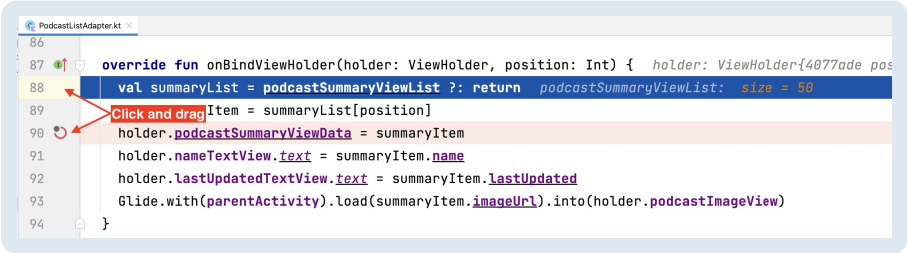
Start searching by tapping the search icon in the app bar and inserting a podcast name. After you confirm the search term, your execution will break successfully:



Code Conditional Breakpoints

Breakpoints can be configured to suspend code only if a specific code condition is met. This is useful if you're trying to find a bug that only appears under certain conditions.

To demonstrate this function, go back to your `onBindViewHolder` dependent breakpoint in **PodcastListAdapter.kt**. Move it down two lines by clicking and dragging it from its current line:

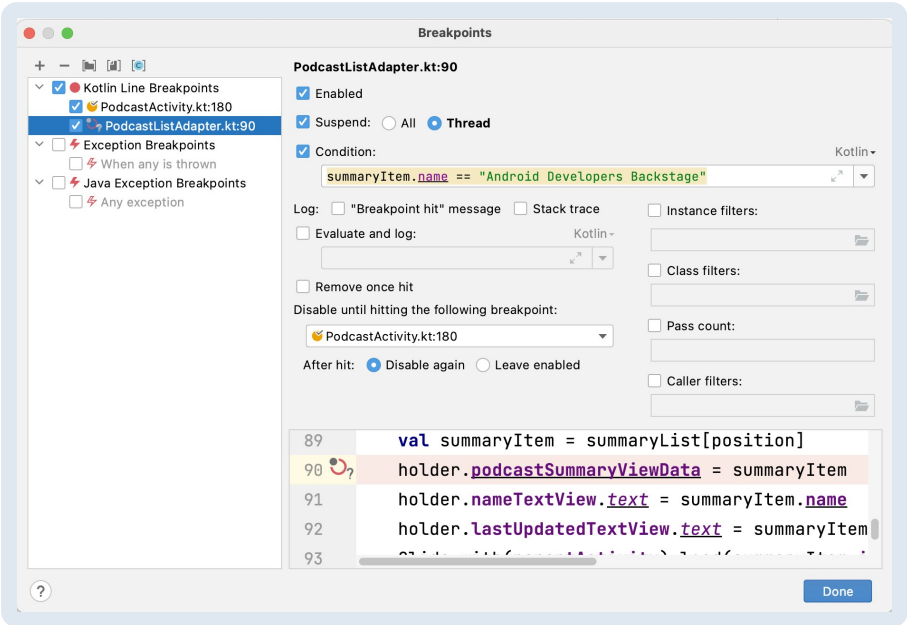


You set the new position after the `summaryItem` variable has been populated, so you'll be able to use that data in a breakpoint code condition.

Open the settings of your breakpoint by right-clicking it and then clicking **More**.

Check the box that says **Condition:** and in the input label underneath, type in this code:

```
summaryItem.name == "Android Developers Backstage"
```



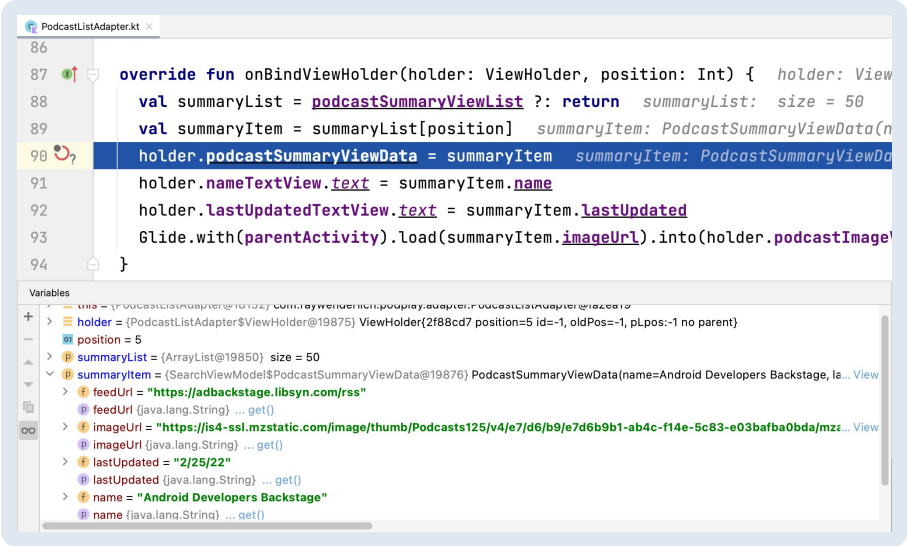
Click **Done** and then resume debugging by clicking **Play**.

Now, your code won't suspend unless both the dependent breakpoint and code

conditions are met. So it should only suspend if you are in the search view and a podcast whose name equals *Android Developers Backstage* is being populated by the adapter.

Try searching a few search terms to verify that the app code does not pause execution; then input “Android” into the search field.

Your code suspends as the adapter is attempting to populate the list item for the podcast Android Developers Backstage:



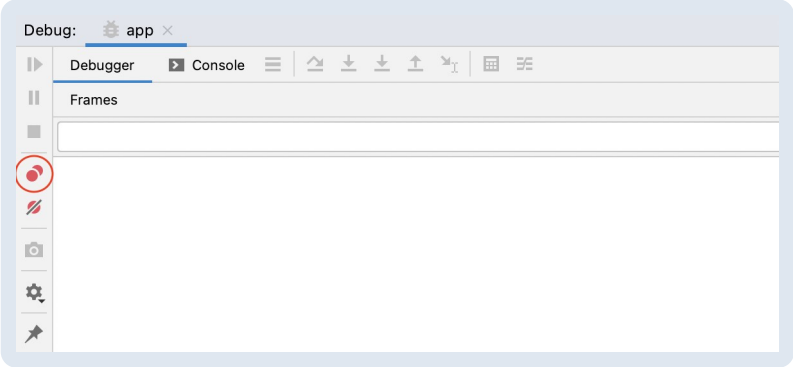
When debugging adapters with many results, a code conditional breakpoint such as this can save a lot of time!

Breakpoints Window

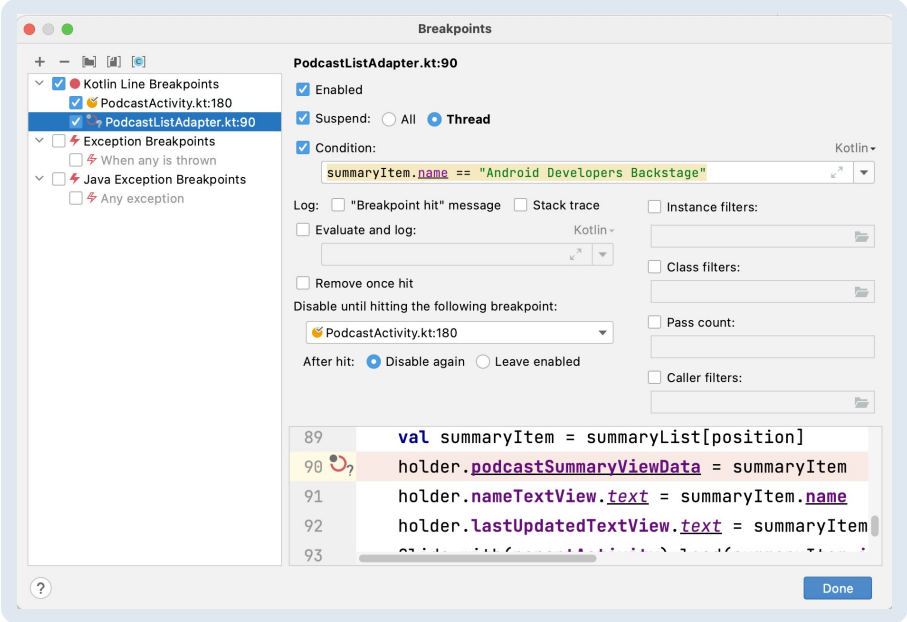
Most engineers set and remove breakpoints as they need them, but a better way is to manage a more extensive set of breakpoints using the **Breakpoints window**.

You can access the Breakpoints window in three ways:

1. Using Android Studio Toolbar - You can the Breakpoints window by clicking **Run ▸ View Breakpoints...**.
2. Right-clicking an existing breakpoint to access its settings and selecting **More**.
3. Clicking **View Breakpoints...** from the Debug window:



The Breakpoints window is a hub for all of your programs breakpoints; you can go to any that you have set, update conditions and enable or disable them all from this window.

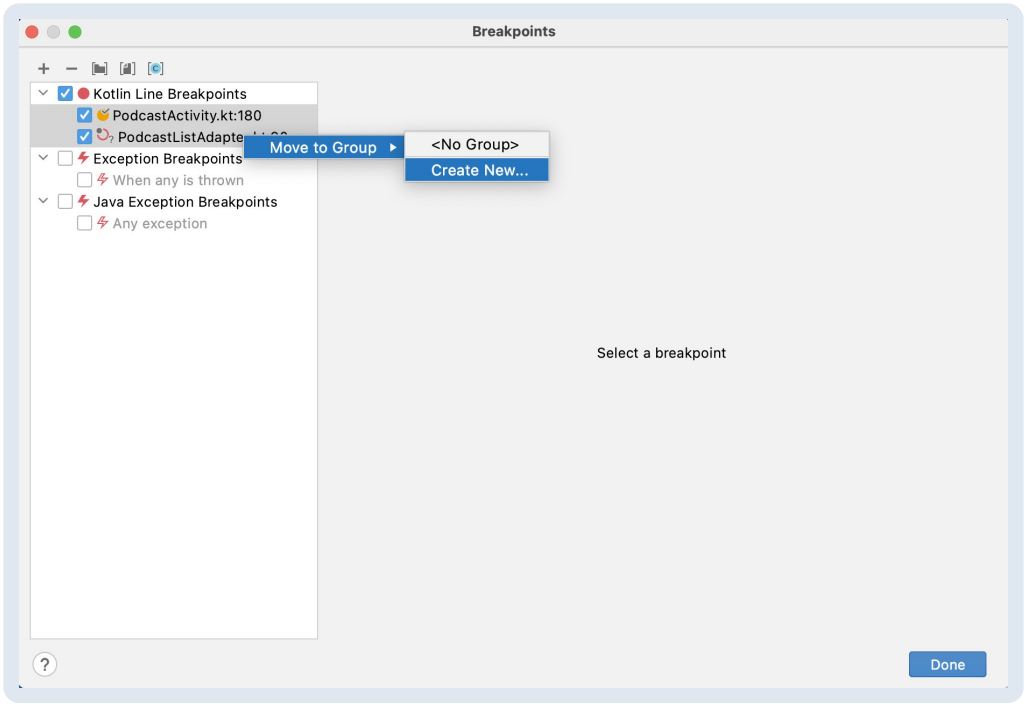


Within the Breakpoints window, you can create groups to store a certain set of breakpoints. These groups can then be disabled and enabled when required.

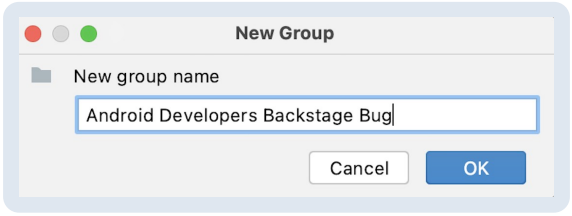
As a scenario, pretend that the Android Developers Backstage podcast had a bug that you were trying to find and fix. A group would be ideal for storing all of the breakpoints related to this bug.

Open the **Breakpoints window** by selecting **Run ▸ View Breakpoints**.

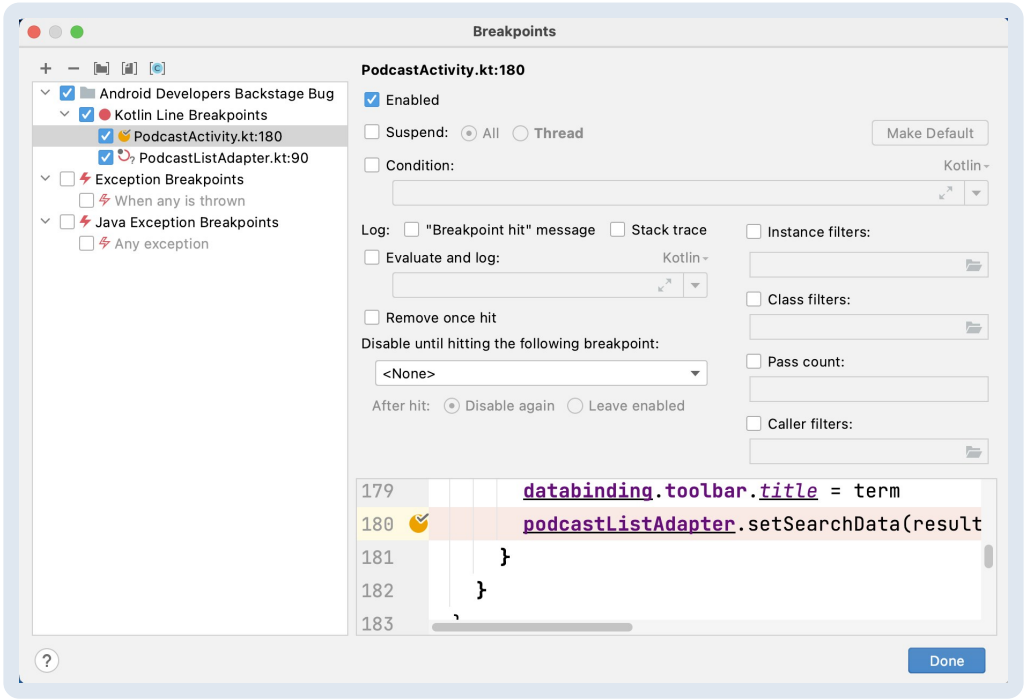
Shift-click on both your `PodcastListAdapter` and `PodcastActivity` breakpoints to highlight them both. Then, right-click them and select **Move to Group ▸ Create New...**.



Type in the name “Android Developers Backstage Bug” and click **OK**.



Your breakpoints have now moved to the group of **Android Developers Backstage Bug** in the Breakpoints window:



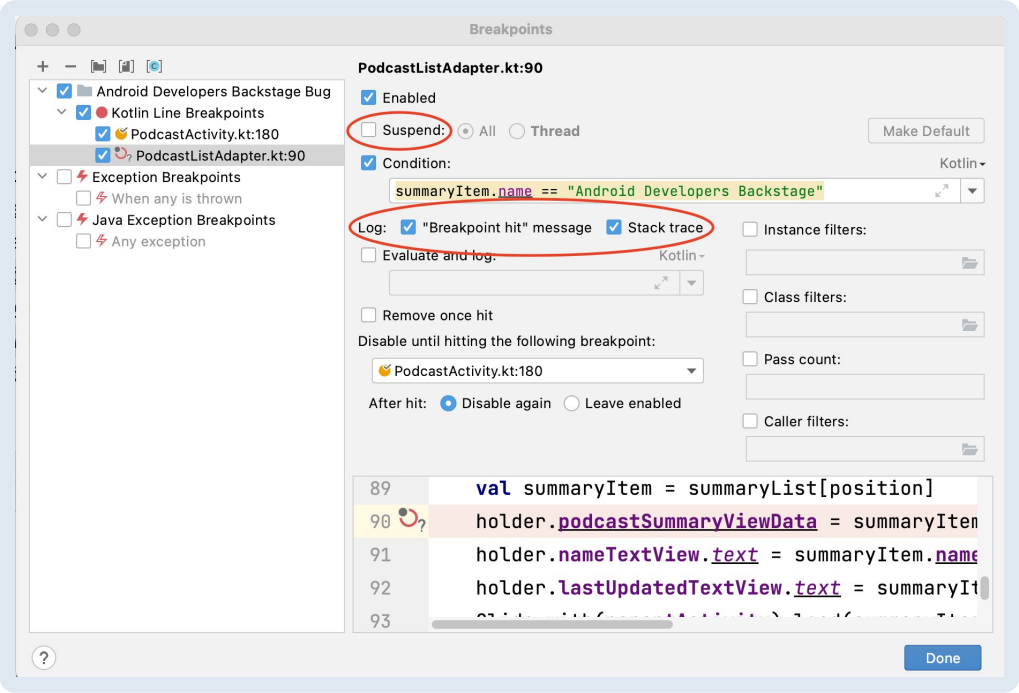
You can toggle these breakpoints on and off by clicking the checkbox next to the

new group. This allows you to continue to work on other activities within your project and come back to bugs at a later time without having to reset all of the necessary breakpoints.

Using Breakpoints to Log Information

Breakpoints don't have to result in a paused app; you can use them to log debug information with the power of unsuspended breakpoints. Sending logs from breakpoints can save you from cluttering your program with temporary log statements. Logging from breakpoints also gives the added benefit of being able to disable and re-enable them at will.

In the **Breakpoints window**, select your **PodcastListAdapter breakpoint**. Uncheck **Suspend** and check **"Breakpoint hit" message** and **Stack trace**.



Your breakpoint will now never suspend the code execution as you've disabled its suspend function; instead, each time it's hit, a "Breakpoint hit" log will be posted to the debug window, as well as the stack trace.

Click **Done** and run your app in debug mode.

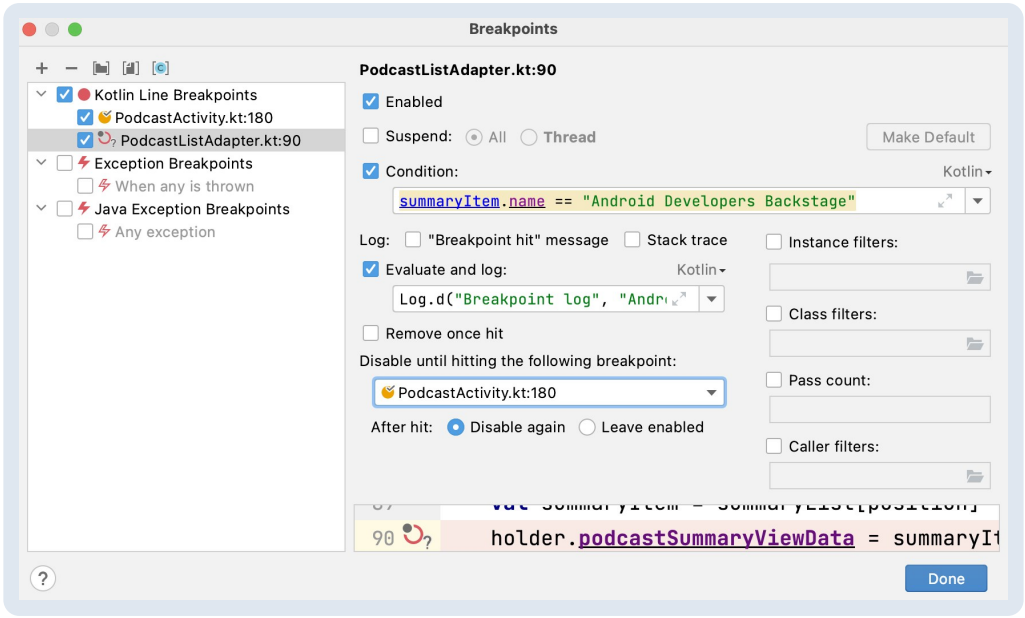
Search "Android Backstage" in the app by tapping the search icon and then clicking the **Debug** tab to see the logs that have been emitted:

You also have the power to log any information that you like from breakpoints, not just the stack trace.

Go back to your **Breakpoints window** and for the **PodcastListAdapter** breakpoint, un-check **"Breakpoint hit" message** and **Stack trace**. Now check **Evaluate and log:** and input this code:

```
Log.d("Breakpoint log", "Android Developers Backstage podcast
loading, feedUrl: " + summaryItem.feedUrl)
```

As long as your breakpoint has access to the variables needed in its scope, you can input any type of log code that you need into the **Evaluate and log** field, logging as little or as much as required. In this case, the breakpoint is logging some useful information, and the podcasts feed URL.

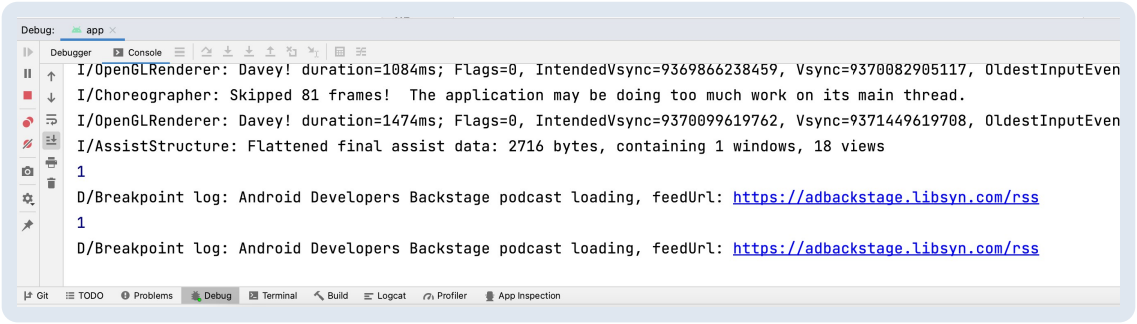


You'll be prompted to add an import for `Log`. Go ahead and follow the tooltip prompt to add the import.

> **Note**: If you don't see the tooltip prompt, set your cursor to `Log` and press **Option-Return** to add the import.

Run your app in debug mode and search for "Android Backstage". Your **Debug**

tab will now show your custom log message when it's hit:



Key Points

- Navigate a suspended app by stepping through code with Step Over, Step Into and Step Out.
- Use the Frames pane to navigate to different scopes within your current execution stack.
- Use Drop Frame to go back in time to a previously executed stack.
- Use breakpoint conditions such as depends on and code conditions to control when a breakpoint should register that it's been hit.
- Use unsuspended breakpoints for breakpoints that shouldn't pause the code execution.
- Place breakpoints in groups so they can easily be enabled/disabled.
- Breakpoints can log information to the debug console.

Where to Go From Here?

You're now a breakpoint genius! No longer will you have to rely on in-program logs and tedious trial and error to locate a bug; with **unsuspended breakpoints**, **breakpoint logging**, and **breakpoint conditions**, you can now use logic to squash those bugs faster.

You might be surprised to hear that there are many additional advanced features that breakpoints have that go above and beyond the average debugging requirements of an engineer and can handle the trickiest of bugs; these include functions such as:

- **Class filters** that limit a breakpoints' operation to particular classes.
- **Caller filters** that limit a breakpoints' operation depending on the caller of the current method.
- **Pass count** that enables a breakpoint only after it's been hit a certain

number of times.

- **Method**, **field** and **exception** breakpoint types that can be used in addition to **line** breakpoints.

Read about these advanced features and even more in the excellent IntelliJ documentation for Breakpoints.

3 Logcat Navigation & Customization

Written by Vincenzo Guzzi

The **Logcat** window in Android Studio acts as a central hub for displaying system and custom messages of applications across Android devices connected via the Android Debug Bridge - ADB.

With the power of the Logcat window, you can view messages in real-time and historically, allowing you to understand exactly how your apps function by viewing the logs they emit.

In this chapter, you'll discover how to utilize the Logcat window to its fullest potential as you navigate PodPlay and fix a selection of bugs by emitting and reading logs. You'll learn how to:

- Navigate the Logcat window.
- Send custom log messages from your app.
- Customize the Logcat window, making it easier to read.
- Filter your logs.
- Use logs to help you find and fix bugs.

Breaking Down the Logcat Window

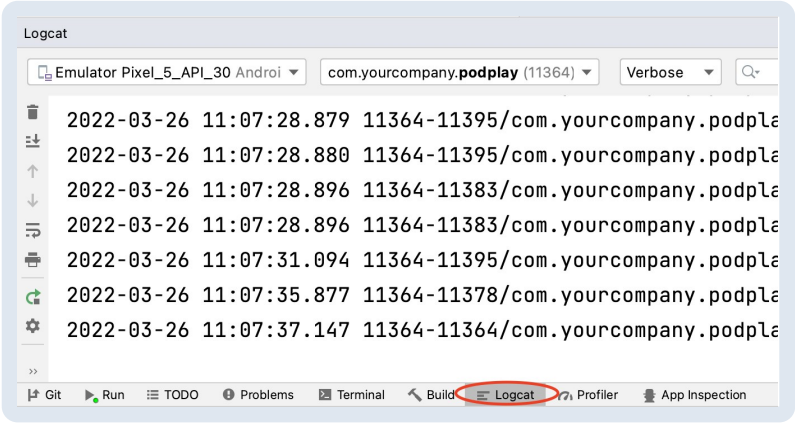
To become familiar with the Logcat window, you'll have to produce some logs first.

Open the **Podplay** starter project in Android Studio and run PodPlay. Complete these actions inside the app:

1. Search for the term **Farm** using the **search** icon.
2. Tap the first podcast in the list of results.
3. Choose the first podcast episode.
4. Tap the **play** icon and wait for it to load and start playing.
5. Pause the podcast.

In Android Studio, expand the Logcat window by clicking **View ▸ Tool Windows ▸ Logcat**.

You can also expand and minimize the Logcat window by clicking the **Logcat** tab at the bottom of Android Studio:



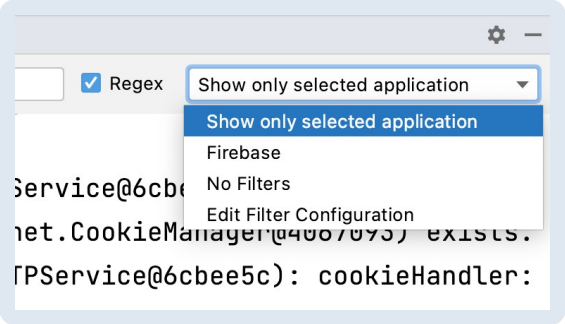
There's now a whole bunch of logs that your Android device has produced; it can be pretty daunting at first sight as there are so many! You're going to work through it bit by bit.

Now take a closer look:

At the top, you can find:

1. **Android device selector** - If you have multiple Android devices connected via the ADB, you can select which device's logs to view with this option.
2. **Process selector** - You can switch between multiple apps or processes with this option. This is useful in cases where you'd like to see how your app interacts with a different Android service such as Bluetooth or the phone's photo gallery app.
3. **Log level** - A quick filter that toggles between different log severity levels.
4. **Filter search bar** - This allows filtering the logs displayed based on text or regex.
5. **Filter menu** - Use this to filter the logs based on the source of executing process.

For now, make sure that you have **Show only selected application** in the filter menu:

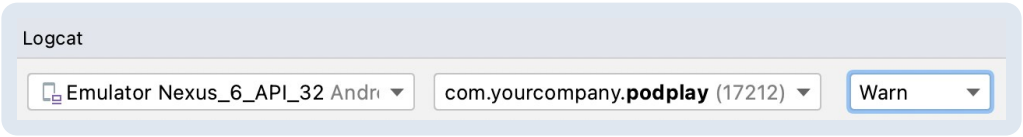


Your Android device will emit all of your system logs if you don't have this selected, even if your active app isn't producing them. In most cases, you'll want this filter on.

Logs that are output by an Android application can be of six different severity levels:

1. **Verbose** - The lowest severity level log; this shouldn't compile these logs into an application outside of development; choose this filter if you want to see *every* log level.
2. **Debug** - Logs that are useful only during development; they should be stripped from release builds.
3. **Info** - General informative log message for standard usage, these are intended to be kept in a release environment.
4. **Warn** - Shows possible issues that could become errors; these are always kept in a released app build.
5. **Error** - Logs that emit due to errors in the application; these will be displayed in a release build.
6. **Assert** - Logs that should never be shown; if they are, something has gone critically wrong with the app.

Next, ensure that the **process selector** is set to **com.yourcompany.podplay** and the **log level** is set to **Warn**.



Setting the log level filter like this tells Logcat only to display logs that are the level of **Warn** or higher. (The log levels higher than **Warn** are **Error** and **Assert**.)

You'll now see several **Warn** level log messages:

```
W/company.podpla: Redefining intrinsic method boolean java.lang.Thread.interrupted(). This may
W/company.podpla: Accessing hidden method Landroid/view/View;->computeFitSystemWindows(Landroid
W/company.podpla: Accessing hidden method Landroid/view/ViewGroup;->makeOptionalFitsSystemWindo
W/System: A resource failed to call close.
W/company.podpla: Accessing hidden method Ljava/lang/invoke/MethodHandles$Lookup;-><init>(Ljava
W/OpenGLRenderer: Failed to choose config with EGL_SWAP_BEHAVIOR_PRESERVED, retrying without...
W/company.podpla: Long monitor contention with owner Binder:11364_1 (11382) at int android.medi
```

Logs that are warnings generally appear as an app launches. This is due to the nature of warnings. A program compiles anything that the program doesn't think is set up perfectly and may cause a future error only once.

In this particular case, the emitted logs are due to the selected emulator and outdated code it uses within the Android open source project libraries, you can ignore them. With Android updates, most of the existing warnings disappear, and new ones may show up. It takes a keen eye to spot a warning that's due to your own code; you should fix these as soon as they're spotted.

Take a detailed look at one log:

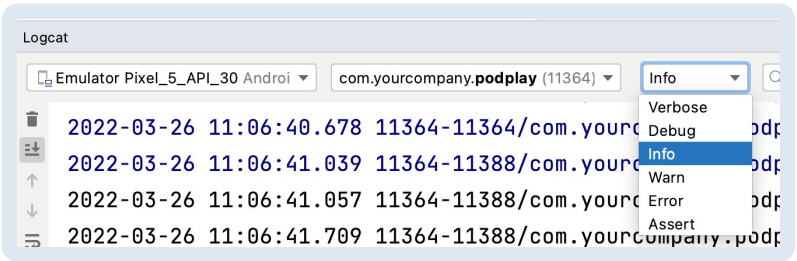
```
2022-03-26 11:06:38.809 11364-11364/com.yourcompany.podplay W/company.podpla: Redefining intrinsic meth
2022-03-26 11:06:40.652 11364-11364/com.yourcompany.podplay W/company.podpla: Accessing hidden method L
2022-03-26 11:06:40.652 11364-11364/com.yourcompany.podplay W/company.podpla: Accessing hidden method L
```

Here's an explanation of the format:

1. The date and time that the log was sent.
2. The identifier of the process and thread that sent the log.
3. Package ID of the app that sent the log.
4. Log priority identifier, e.g. **W** for **Warn**, **I** for **Info** etc.
5. The log tag that helps identify where the log came from; usually the class name.
6. The log message itself appears last.

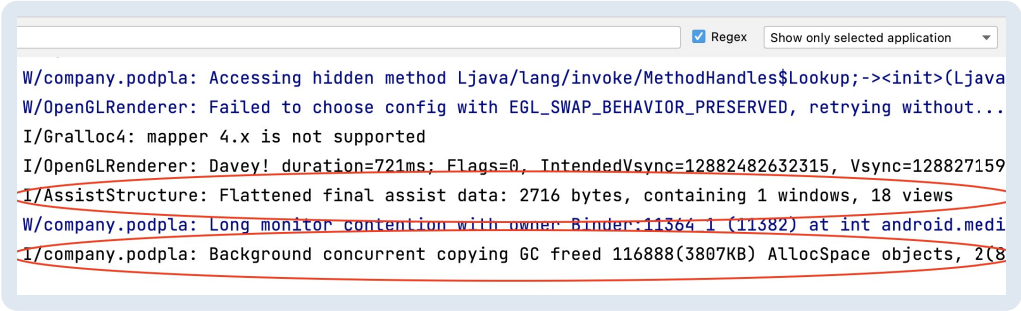
This is the format for every log, no matter the log level.

Now, click the **Log level** drop-down and select **Info**:



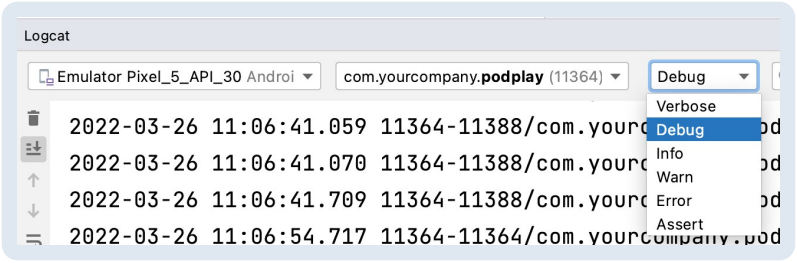
With the log severity level decreased, you'll see some additional **Info** logs that

have appeared in the Logcat window. These are general information logs that tell you useful information about your running processes.

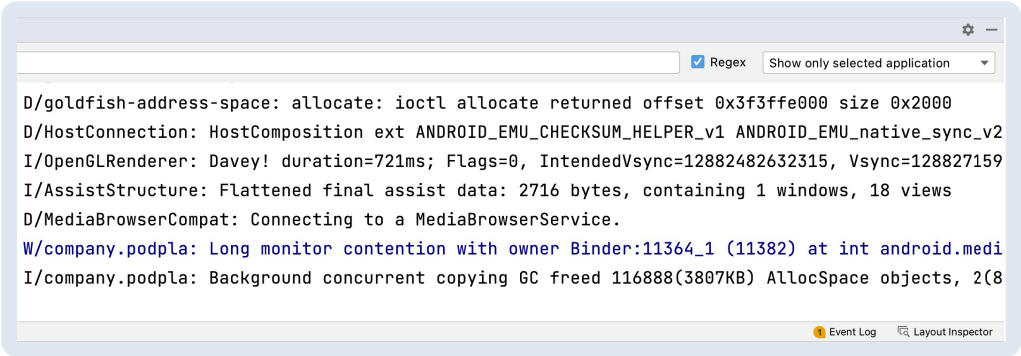


In this example, there are new **Info** logs telling you that the Java garbage collector has freed up some memory and another that tells you the number of windows and views displayed.

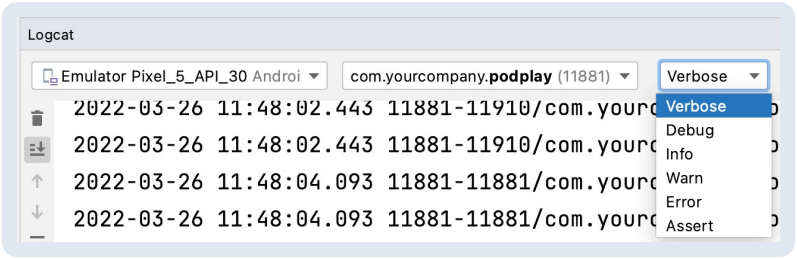
Go back to the **Log level** drop-down again and select the **Debug** filter:



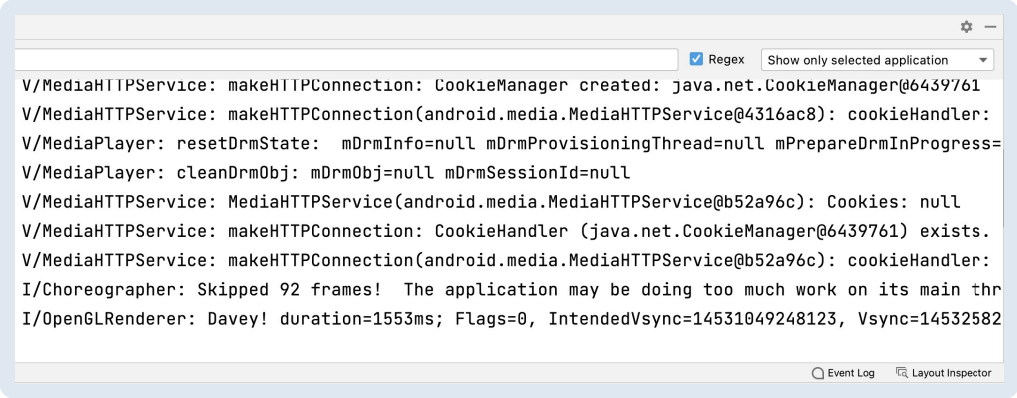
You can see debug logs, such as “Connecting to a MediaBrowserService” and other, more detailed logs that can help you debug the app:



Finally, enable all of the logs by selecting **Verbose** in the **Log level** drop-down:



The Logcat fills up with a *lot* more logs with the **Verbose** level enabled. You can see logs detailing things like the app’s API connections:



Now that you've enabled all the logs, you're able to see a lot of system information that tells you what the app is doing at each moment.

However, this information is pretty useless if you're not looking at the logs in real-time, as you have no way of linking the system logs to what actions are being completed within the app. This is because you're not sending any **custom logs**.

Next, you'll go through the process of adding custom logs throughout the PodPlay code so that you can easily interpret what PodPlay is doing at each moment from the Logcat.

Adding Custom Logs

You can create and post any kind of log message to Logcat. Doing so can be a powerful tool that can help you debug your app without needing to set breakpoints. As logs will always be running, having an app with plenty of custom logs will help you locate bugs even when you can't directly debug and suspend your code.

To send logs to the Logcat, there is a utility class in Android named Log. You'll use this class to set your custom log severity levels, tags, and messages.

In the starter project, open **PodcastActivity.kt** and add the **Log import** to the top of the file next to all of your other imports:

```
import android.util.Log
```

Above the `PodcastActivity` class declaration, add a new constant variable called `TAG` and set the value to **PodcastActivity**:

```
private const val TAG = "PodcastActivity"
```

This will be the tag you'll use for all of the **custom logs** that you'll send within `PodcastActivity`. You can, of course, set tags to anything. As a default, it's best practice to have your custom log tag as the class which has sent the log so you can easily map the location.

Info Logs

When you're viewing logs, the main piece of information that you need to know is where the app code is during an event. To do this, you can decorate your code with **Info logs** that simply state what has been called.

Underneath `onCreate()` inside `PodcastActivity`, add an override for `onResume()` and add a **custom log** inside the method that tells the Logcat that you've called this method:

```
override fun onResume() {
  super.onResume()
  Log.i(TAG, "onResume() called.")
}
```

The `Log` class has a number of different static methods which dictate the logs severity level, the very same severity levels discussed earlier in this chapter:

- **Log.v()**: Verbose.
- **Log.d()**: Debug.
- **Log.i()**: Info.
- **Log.w()**: Warning.
- **Log.e()**: Error.

Each method takes a **tag** and **message** `String` parameter as well as an optional **Throwable** parameter that logs an exception.

Scroll further down in `PodcastActivity` until you get to `onNewIntent()`. Replace `// TODO 1` in this method with another info log:

```
Log.i(TAG, "onNewIntent() called with intent: ${intent.action}.")
```

Here, you are again logging the information that you've called `onNewIntent()`. You're also passing the action of the `Intent` within your log's message string. It's always best to include any relevant information in your logs that may be able to help with debugging your code.

Below, find `onShowDetails()` and replace `// TODO 2` with a custom log:

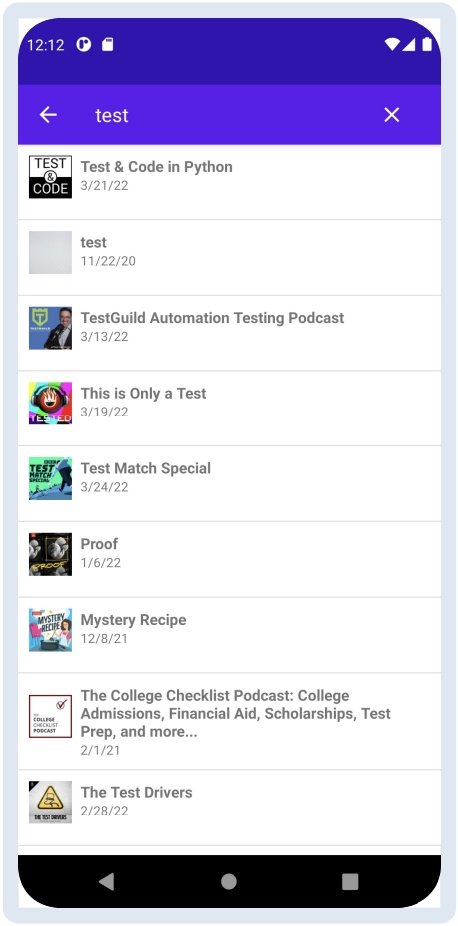
```
Log.i(TAG, "Retrieving podcast feed for podcast named
'${podcastSummaryViewData.name}'")
```

You're leaving more detailed information that you're about to retrieve a podcast feed.

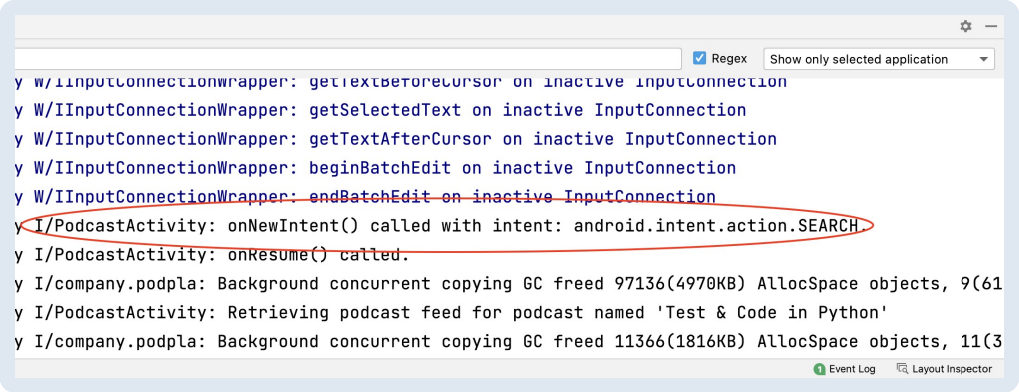
Now, build and run the app again, ensuring you have the Logcat window expanded. Switch the severity level filter to **Info**. You only want to see the logs of severity level **Info** and higher right now.

You'll see right away that you're now getting an "onResume() called" log appear in the Logcat window:

In PodPlay, tap on the search icon and enter the term **test**:



Tap on the first podcast that you see and observe the logs:



You'll now see it sending additional custom logs that inform you're calling `onNewIntent()` and that it is retrieving a podcast feed.

With only sending these three additional logs, the Logcat window has become a lot easier to read as, in between the system logs, there are details of what the app is doing at any given time.

Debug Logs

Now that you've added some Info logs and seen the clarity that they can add to the Logcat, you'll gain even more insight by adding debug logs that will inform the Logcat reader of the speed at which it can retrieve a podcast's information.

Open **PodcastViewModel.kt** and, just like previously, add a constant variable called **TAG** above the class declaration:

```
private const val TAG = "PodcastViewModel"
```

Also, add the Log utility method import at the top of the file:

```
import android.util.Log
```

Now, scroll down to the method `getPodcast()` and replace `// TODO 3` with a new **custom log**:

```
Log.d(TAG, "Calling getPodcast for podcast named
${podcastSummaryViewData.name}, recording time.")
```

You're setting this log's severity level to **Debug** as you'll use this to debug how long it takes to retrieve a podcast from the repository. This log states the podcast name that the app is about to retrieve data for; it also states that it's recording the time.

Next, you'll need to measure the actual time to log this in a message when the call is complete. You'll use the Kotlin system method **measureTimeMillis()** to do this.

To be able to use the method, add it to the **imports** section at the top of the file:

```
import kotlin.system.measureTimeMillis
```

Now, go back to `getPodcast()`. Find `// TODO 4` and replace the `getPodcast()` call wrapped in `measureTimeMillis()`:

```
val timeInMillis = measureTimeMillis {
  podcast = podcastRepo?.getPodcast(url)
}
```

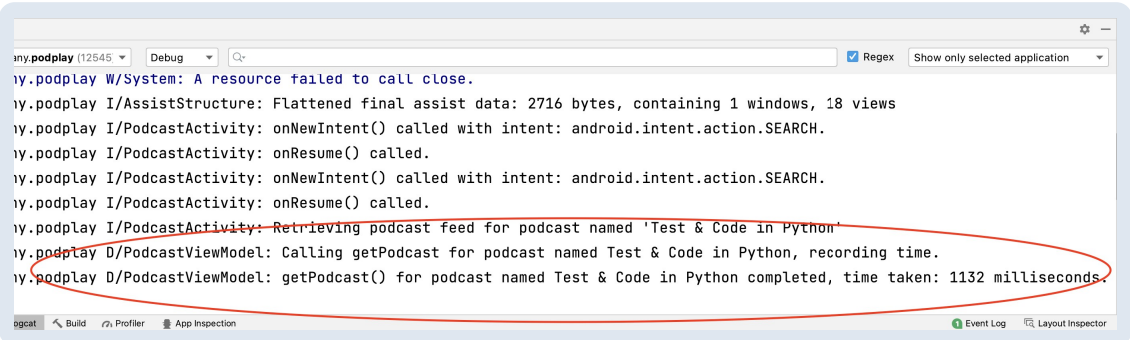
`timeInMillis` is now populated with the number of milliseconds that it took to execute the `getPodcast()` repository call. Create a new custom log that shares this information by replacing `// TODO 5` further down in the method with:

```
Log.d(TAG, "getPodcast() for podcast named
${podcastSummaryViewData.name} completed, time taken: $timeInMillis
milliseconds.")
```

Switch the Logcat severity level to **Debug**, then build and run the app to see this new log in action.

Search for a podcast with the term **test** by tapping on the search icon and then tap on the first podcast in the results list.

You'll receive some awesome debug information that details how long it takes the app to retrieve a podcast's details:

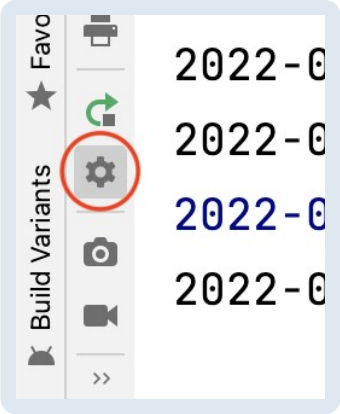


Filtering and Customizing the Logcat Window

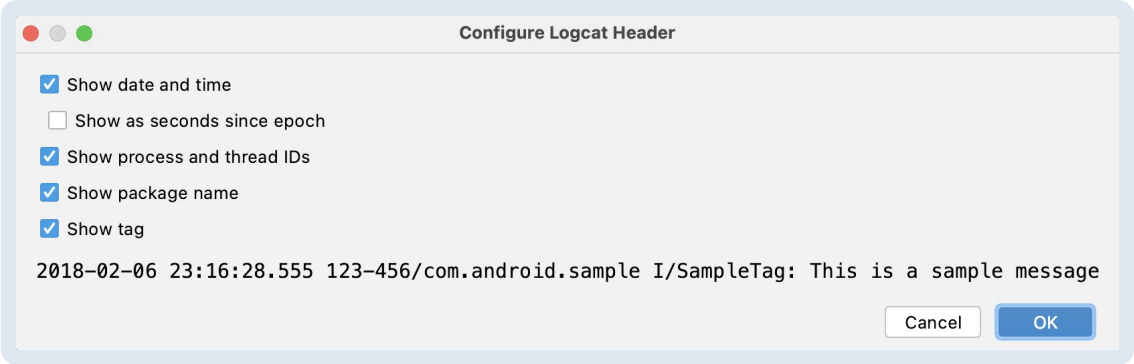
When the Logcat window emits many logs, it can become quite cumbersome; it's hard to navigate and locate the logs you care about quickly. There are ways to improve this with customization and additional filtering.

Customizing Logcat Headers

In the Logcat toolbar to the left, there is a cog icon that has the label **Logcat Header**:

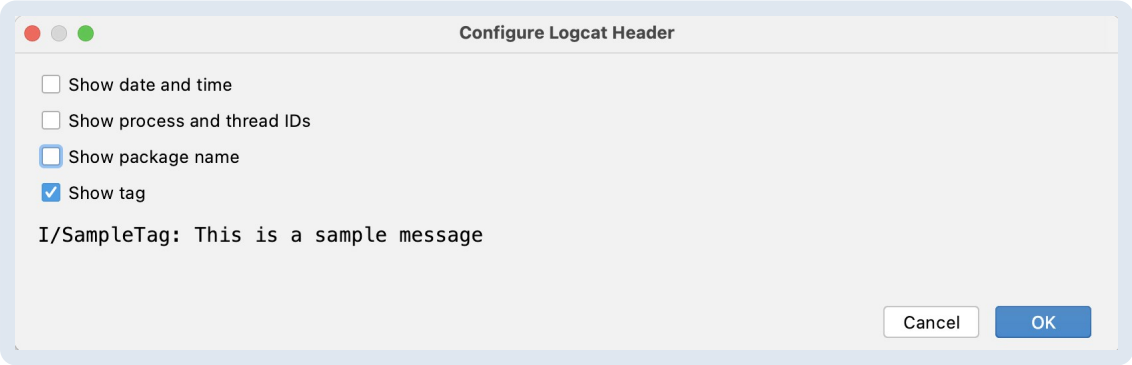


Click this cog and a **Configure Logcat Header** dialog will appear:



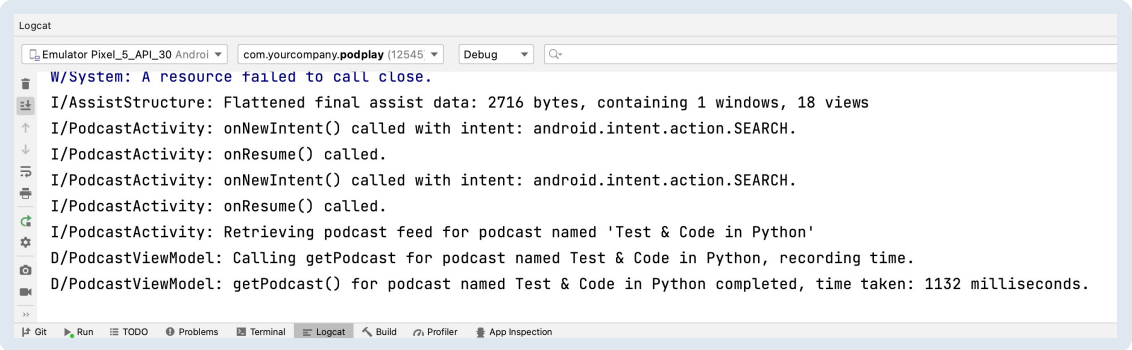
You can choose what information you'd like to include in each log using this dialog.

Go ahead and uncheck **Show date and time**, **Show process and thread IDs** and **Show package name**:



Click **OK**.

Your logs now won't have all of the prefixed metadata, like package and thread ID, making the logs simpler and easier to read.



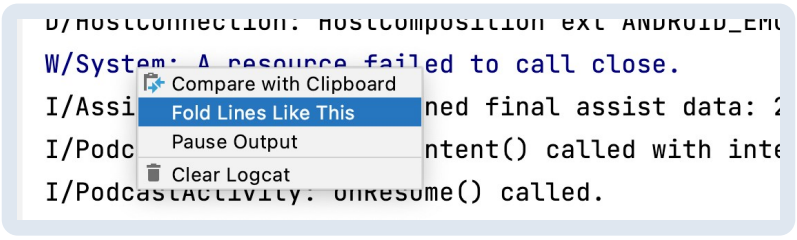
> **Note**: You can re-enable any prefixes by going back to the **Logcat Header** setting and checking the relevant boxes.

Folding Logs

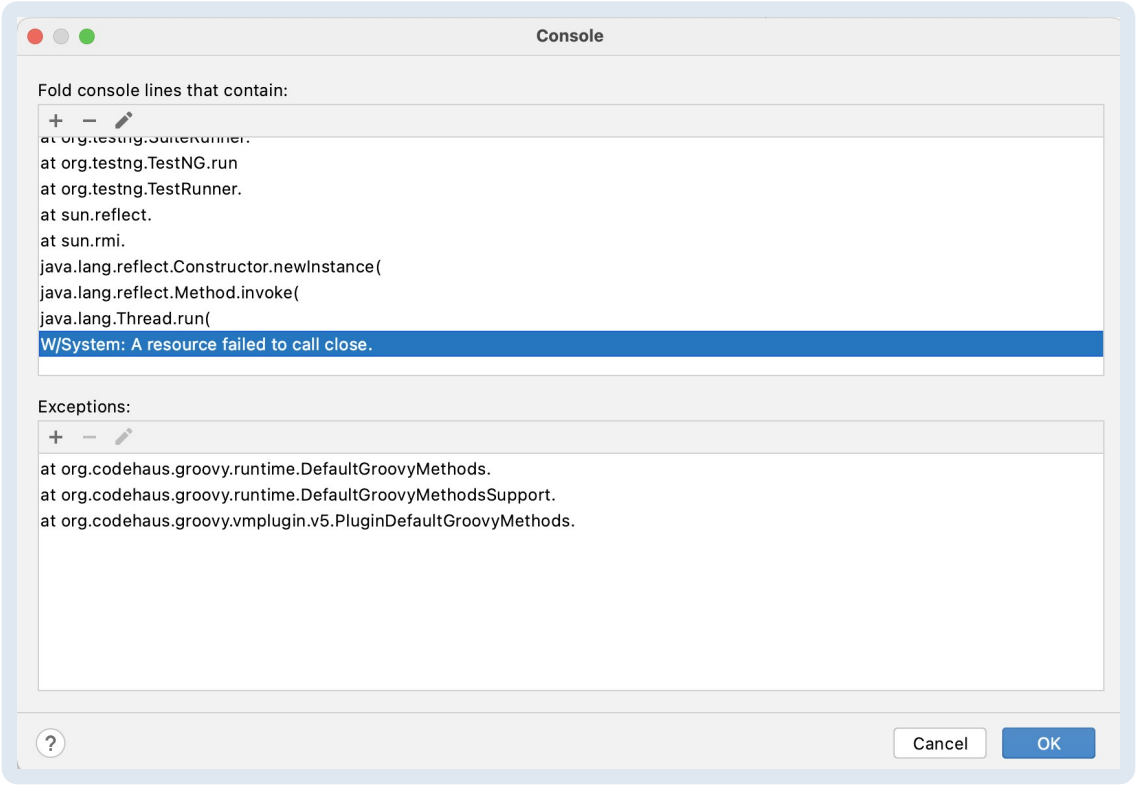
Folding is another tool that you can use to partially remove logs that aren't important to you from view. It's collapsing some data into a condensed view, so it takes up less screen real estate; you can then usually unfold or expand the collapsed view with a click.

You can use this method of folding with Android code by collapsing scopes like `for` loops and `if` statements; you can even collapse whole classes.

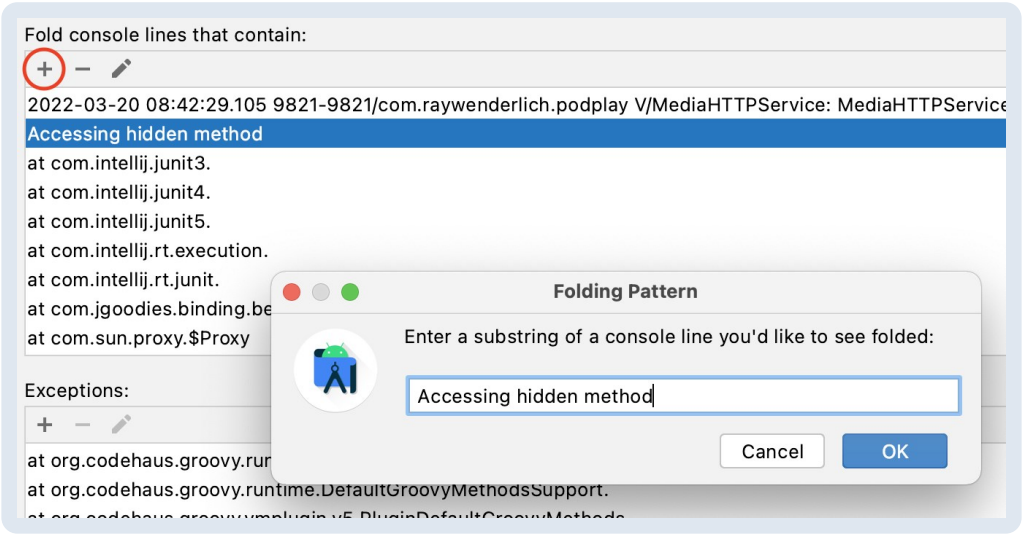
You can do this with logs too! Find a log that you would rather fold from view, right-click and select **Fold Lines Like This**:



This will open a console dialog where you can add, remove or edit the regex pattern of logs you want to fold:



You can add any type of text to this list. If it matches the results of a log, it will fold. To demonstrate this, input a new line by clicking the plus icon and entering the text **Accessing hidden method**.



Click **OK**.

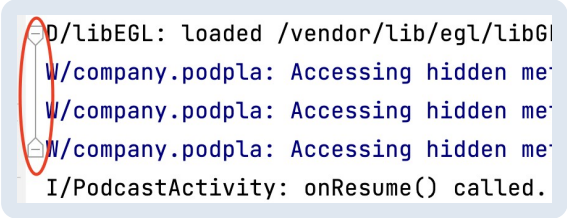
With the **Verbose** severity level set, scroll up in the Logcat window, and you'll see folded logs that state **<3 internal lines>**:

```
D/libEGL: loaded /vendor/lib/egl/libGLESv1_CM_emulation.so
D/libEGL: loaded /vendor/lib/egl/libGLESv2_emulation.so <3 internal lines>
I/PodcastActivity: onResume() called.
```

Clicking this will expand the folded logs:

```
D/libEGL: loaded /vendor/lib/egl/libGLESv2_emulation.so
W/company.podpla: Accessing hidden method Landroid/view/View;->computeFitSystemWindows(Landroid/
W/company.podpla: Accessing hidden method Landroid/view/ViewGroup;->makeOptionalFitsSystemWindow
W/company.podpla: Accessing hidden method Ljava/lang/invoke/MethodHandles$Lookup;-><init>(Ljava/
I/PodcastActivity: onResume() called.
```

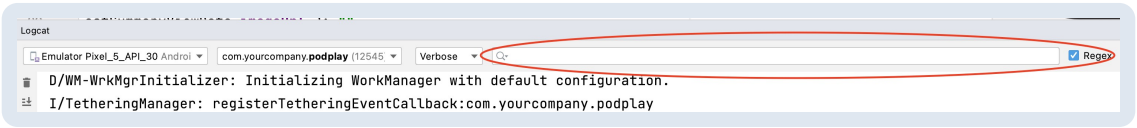
Clicking the fold icon to the left of these logs will collapse them again:



By setting up your fold filters appropriately, you can get to a state within the Logcat window where only logs you would actively like to view begin in an expanded state, whereas all others are folded.

Creating Custom Filter Configurations

The easiest filter to use is also the most powerful, the **Logcat search bar**:



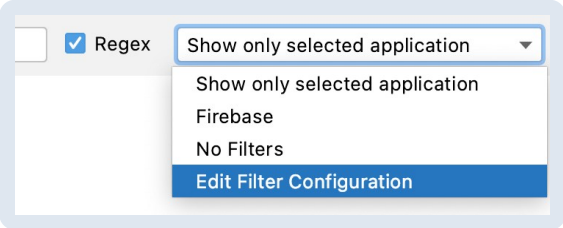
The search bar can match exact text for the logs you'd like to find, with the ability even to match regex patterns for more advanced searching.

Every part of a log is searchable, including the tag. Searching for tags can be a powerful and quick way to find the logs you're looking for instantly. Add **PodcastActivity** into the search bar to quickly find all your logs with that tag that have:

You may want to save a search that you find yourself repeatedly using so that you can quickly select it. You can do this with **Filter Configurations** located to the right of the search bar.

Click **Edit Filter Configurations** by selecting the **Filter Configurations** drop-

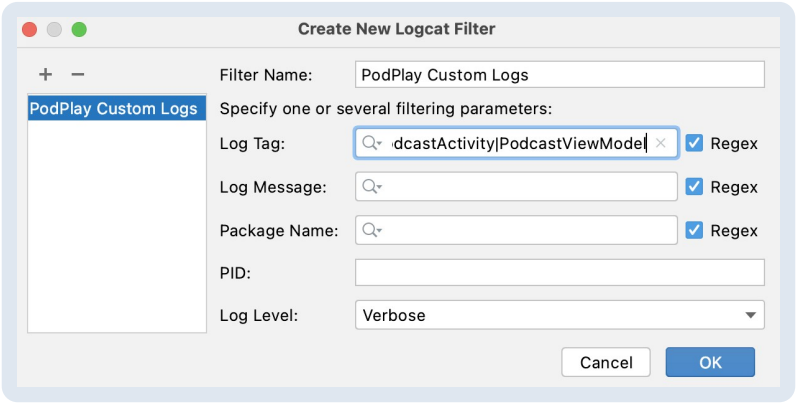
down:



In **Edit Filter Configurations**, you have access to an even more powerful set of search settings that allow you to combine multiple search results into one filter.

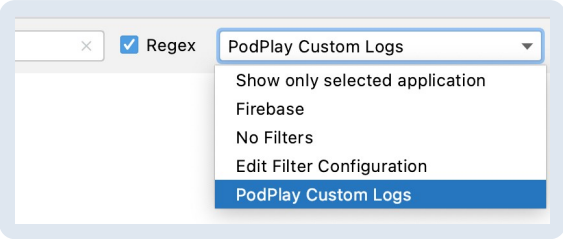
Add a permanent filter configuration for viewing your custom logs so you can easily toggle it to on in the Logcat window.

In the **Filter Name** input field type **PodPlay Custom Logs**, and in the **Log Tag** input field add the regex **PodcastActivity|PodcastViewModel**:



Click **OK**.

This will create a new filter that searches for all logs that have the tag of either **PodcastActivity** or **PodcastViewModel**. You can select this filter, or a different one, by going back to the **Filter Configurations** drop-down:

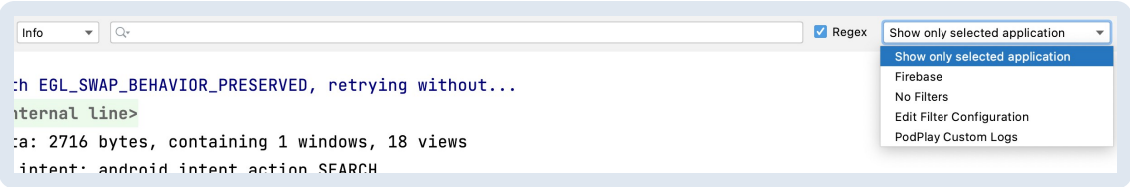


Using Logs to Fix a Bug

Now that you've mastered the art of logging, you'll put these new skills into practice by identifying a bug and then fixing it.

Your first step is to find a bug.

Launch PodPlay and make sure that you have the Logcat window open to see emitting logs while navigating the app. Set the **Log level** filter to **Info** and make sure you're not using the custom filter configuration:



Search for a podcast by inputting **Science** into the search window of PodPlay and tap on the first podcast in the list of results. Then, select the first podcast episode. You'll now be on an episode with the ability to play the podcast.

Press the **Play** icon and observe the lag between pressing play and the podcast actually playing.

This is strange behavior; something is clearly misbehaving, the Logcat can help identify the problem.

Open the Logcat window, and you'll see that an **Info** log with the tag of **Choreographer** has been output:

The message says:

> "Skipped 42 frames! The application may be doing too much work on its main thread."

This log appears exactly as you press the **Play** icon for a podcast episode but not on subsequent playing after pausing the episode.

Interesting!

Armed with this information, you know that something is wrong; something is blocking the main thread and causing lag.

The next step is to add custom logs to `PodplayMediaCallback` so you can locate the exact method that is blocking the main thread.

Using Custom Logs to Locate a Problem

Open **PodplayMediaCallback.kt**. Above the class declaration add a constant called **TAG**:

```
private const val TAG = "PodplayMediaCallback"
```

Now add the **Log** utility import to the top of the file so you can use it throughout this class:

```
import android.util.Log
```

To locate the offending method, you'll need to add custom logs that inform you of when it calls each method to get more clarity on what PodPlay is doing when the bug occurs.

Start off with `onPlay()`.

Scroll down `PodplayMediaCallback` until you find `onPlay()` and replace `// TODO 6` with an **Info** log:

```
Log.i(TAG, "onPlay() called.")
```

Do the same in `onStop()` by replacing `// TODO 7`:

```
Log.i(TAG, "onStop() called.")
```

Finally, modify `onPause()` by replacing `// TODO 8`:

```
Log.i(TAG, "onPause() called.")
```

Scroll down to `initializeMediaPlayer()` and replace `// TODO 9` with an Info log that informs it has been called:

```
Log.i(TAG, "initializeMediaPlayer() called.")
```

Similar to before, replace `// TODO 10` within `prepareMedia()`:

```
Log.i(TAG, "prepareMedia() called.")
```

And finally, inside `startPlaying()` replace `// TODO 11` with another Info log:

```
Log.i(TAG, "startPlaying() called.")
```

You've now added six Info logs to `PodplayMediaCallback`. These will give you a lot more context when trying to locate the bug that's blocking the main thread.

Build and run PodPlay and, once again, navigate to a podcast episode and play it by pressing the **Play** icon.

```
I/PodplayMediaCallback: onPlay() called.
I/PodplayMediaCallback: initializeMediaPlayer() called.
I/PodplayMediaCallback: prepareMedia() called.
I/PodplayMediaCallback: startPlaying() called.
I/Choreographer: Skipped 46 frames!  The application may be doing too much work on its main thread.
I/OpenGLRenderer: Davey! duration=786ms; Flags=0, IntendedVsync=19035332514029, Vsync=19036099180665,
```

Take a look at the logs; you can see that the Logcat is emitting each **Info** log and the skipped frames system log appears at the end. This doesn't tell you much about the exact method causing issues, but you may have noticed that one of the methods took longer to execute.

If you don't see the log times, enable the time Logcat header, **Show date and time**, by clicking the **Configure Logcat Header** settings icon, then look at the logs again:

```
2022-03-26 13:03:08.364 I/PodplayMediaCallback: onPlay() called.
2022-03-26 13:03:08.369 I/PodplayMediaCallback: initializeMediaP
2022-03-26 13:03:08.371 I/PodplayMediaCallback: prepareMedia() c
2022-03-26 13:03:09.121 I/PodplayMediaCallback: startPlaying() c
2022-03-26 13:03:09.143 I/Choreographer: Skipped 46 frames!  The
2022-03-26 13:03:09.154 I/OpenGLRenderer: Davey! duration=786ms;
```

You can see that a larger than average amount of time passed between `prepareMedia()` and `startPlaying()`.

`prepareMedia()` is blocking the main thread!

Fixing the Bug

This book is about locating problems, not necessarily how to fix them. But you can can solve this particular problem by wrapping `prepareMedia()` inside `onPlay()` in an asynchronous call, so it doesn't block the main thread:

```
CoroutineScope(Dispatchers.IO).async {
  prepareMedia()
}.invokeOnCompletion{
  startPlaying()
}
```

This requires adding three additional imports to the top of the file:

```
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.async
```

Now, if you reinstall the app and try playing an episode, you'll notice that the "skipped frames" is no longer displayed, indicating that the main thread is not blocked. You've successfully fixed a bug by observing the Logcat.

Challenge

A frustrating situation happens when you attempt to click the podcast series called **Early Middle Ages**; it doesn't do anything!

The Logcat window is sparse when clicking this podcast; it doesn't log any debug or info logs that state what is wrong. It's up to you to change that in this challenge.

Use what you've learned so far with breakpoints and logging to find out why this bug is happening and implement a viable solution.

Go to the **challenge** or **final** projects for this chapter if you need help locating this bug.

Key Points

- Logs can be of different severity levels: Verbose, Debug, Info, Error, Warn and Assert.
- You can configure the Logcat window to make it easier to read by clicking the Logcat Header icon.
- Fold logs that you don't care about by right-clicking it and selecting Fold Lines Like This.
- Send custom logs from the app using the Util Log method.
- Decorate the app with Info logs so it can debug the Logcat window more easily.
- Add custom filer configurations by clicking Edit Filter Configurations in the Logcat window.

Where to Go From Here?

In this chapter, you learned a lot about logs and how to customize Logcat as per your needs. Great job!

If you're interested in discovering more about this topic, check out the Android documentation about Logcat.

Don't stop here. In the following chapter, you'll find out how to deal with stack traces and how to analyze them.

4 Analyzing the Stack Trace

Written by Zac Lippard

In the previous chapter, you learned about Logcat and how you can use the Logcat window to find bugs in your app. One common example of Logcat is finding stack traces for crashes and exceptions.

Analyzing a stack trace, and understanding how the trace itself is structured, provides clues as to why the app encountered the error, to begin with.

In this chapter, you'll learn how to read a stack trace from the Podplay app, use tools to navigate through it and fix the associated bug behind the trace. You'll be able to:

- Read through a stack trace.
- Catch errors and rethrow them with more information.
- View the associated threads in a stack trace.
- Add Firebase **Crashlytics** to your app.
- Import a Crashlytics stack trace and fix the underlying error.

Defining Stack Trace

A stack trace shows a list of methods called at a certain place in your code. This list is typically referred to as the **stack frame** or **call stack**. In Android development, an exception generates a stack trace. You can review the stack trace to determine the underlying cause of a bug in your app.

For example, you may have the following methods, `a()`, `b()` and `c()`:

```
fun a() {
  b()
}

fun b() {
  c()
}

fun c() {
  throw Exception("Uh oh!")
}
```

When the exception throws, the stack trace will contain the methods with the

most recent method call at the top:

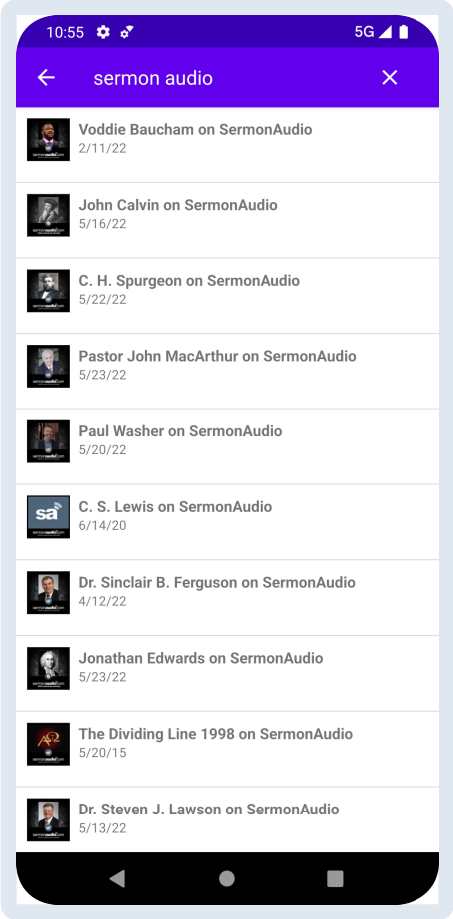
```
c()
b()
a()
```

Stack traces in Android will also include line numbers to denote where the next function call is in the stack.

Next, you'll learn how to read and utilize stack traces to fix a bug in the Podplay app.

Reading a Stack Trace

Open the **Podplay** starter project and run the app. After the app launches, tap the search icon, type in "sermon audio" and press **Return**.



Once the list of podcasts appears, tap the first podcast in the list. The app crashes!

Open the **Logcat** window in Android Studio, switch to the **Error** type, and find a stack trace similar to the following:

```
E/AndroidRuntime: FATAL EXCEPTION: main
    Process: com.yourcompany.podplay, PID: 22519
    java.text.ParseException: Unparseable date: "Fri, 11 Feb 2022
02:10 GMT"
        at java.text.DateFormat.parse(DateFormat.java:362)
        at
com.yourcompany.podplay.util.DateUtils.xmlDateToDate(DateUtils.kt:5
6)
        at
com.yourcompany.podplay.repository.PodcastRepo.rssItemsToEpisodes(P
odcastRepo.kt:75)
        at
com.yourcompany.podplay.repository.PodcastRepo.rssResponseToPodcast
(PodcastRepo.kt:88)
        at
com.yourcompany.podplay.repository.PodcastRepo.getPodcast(PodcastRe
po.kt:61)
        at
com.yourcompany.podplay.repository.PodcastRepo$getPodcast$1.invokeS
uspend(Unknown Source:15)
        at
kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(Cont
inuationImpl.kt:33)
        at
kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:106)
        at android.os.Handler.handleCallback(Handler.java:883)
        at android.os.Handler.dispatchMessage(Handler.java:100)
        at android.os.Looper.loop(Looper.java:214)
        at
android.app.ActivityThread.main(ActivityThread.java:7356)
        at java.lang.reflect.Method.invoke(Native Method)
        at
com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(Runtime
Init.java:492)
        at
com.android.internal.os.ZygoteInit.main(ZygoteInit.java:930)
```

There are a few things to explain:

1. Look at the first line:

   ```
   E/AndroidRuntime: FATAL EXCEPTION: main
   ```

   The `E/AndroidRuntime` part denotes that this was an error log with **AndroidRuntime** as the tag. You'll also see that the exception occurred on the `main` thread.

2. Notice the second line:

   ```
   Process: com.yourcompany.podplay, PID: 22519
   ```

   It provides process-specific information, including the process name, the app's package ID and the process ID.

3. Move to the next line:

```
java.text.ParseException: Unparseable date: "Fri, 11 Feb 2022
02:10 GMT"
```

The third line details what type of exception was thrown, along with an associated message attached to the exception. In the case above, the `ParseException`, was thrown because the date provided isn't parsable.

4. The remaining lines detail the call stack. Each following line represents a method, and the associated line number is provided. The line numbers from the previous method in the stack correlate to the method call above it. For example:

```
at java.text.DateFormat.parse(DateFormat.java:362)
at
com.yourcompany.podplay.util.DateUtils.xmlDateToDate(DateUtils.k
t:56)
```

These lines represent that at line 56 of the `DateUtils` class `xmlDateToDate()` makes a call to `inFormat.parse(date) ?: Date()`.

Knowing where the crash occurs in the code is crucial to fixing the underlying problem. Now that you know what is causing the crash, it's time to fix it!

Catching and Rethrowing Errors

In many cases, the best way to prevent crashes from thrown exceptions is to catch the exception in a try/catch block. The `try` block will run your code, and the `catch` block will catch the exception types you specify. You can use this to handle the crash above.

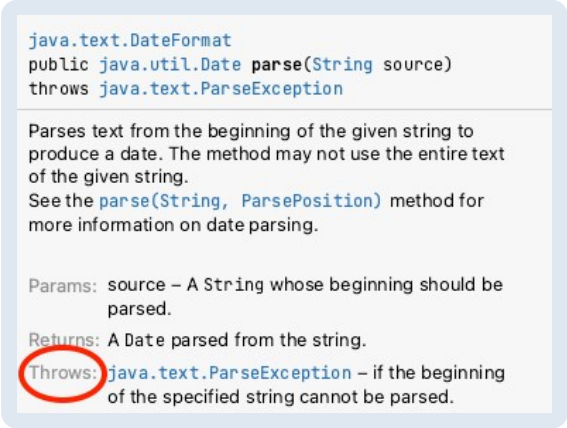
> **Note**: Catching exceptions isn't always necessary. It may make more sense to fix the crash in certain situations. For example, if you encounter a `NullPointerException` you may need to add a `null` check to your code to prevent the offending line of code from being reached if a `null` value is provided.

Open **DateUtils.kt** and find `xmlDateToDate()`. Remember, based on the stack trace line 56 is the culprit:

```
return inFormat.parse(date) ?: Date()
```

On that line, hover your cursor over `parse()`, and the code documentation for that method appears. If the code documentation isn't displayed, move your

cursor to the method and press **F1**. The code documentation explains that the method will throw a `ParseException` when the date formatter can't correctly parse the provided date text.



Now that you know `parse()` can throw an exception, it's time to handle it. Replace line 56 by wrapping the logic inside a try/catch block:

```
return try {
  inFormat.parse(date) ?: Date()
} catch (e: ParseException) {
  Log.wtf("xmlDateToDate", e)
  Date()
}
```

> **Note**: Make sure you use the `java.text.ParseException` import

The `return` statement takes the entire try/catch block. If you can parse the date string, or if `parse()` returns `null`, a `Date` object will be returned in the `try` block.

But, if `parse()` throws a `ParseException`, your code will now catch this. First, the exception will be logged using the `Log.wtf()` (which stands for "What a Terrible Failure" of course!) providing the custom error tag of "xmlDateToDate" and the exception `e`. Then, you set a new `Date` instance as a return value. Providing a `Date` object even in the `catch` block ensures that the app can continue moving forward.

Rerun the app and follow the steps noted earlier to try to reproduce the crash. This time there's no crash!

Open the **Logcat** window in Android Studio, and you'll find the exception:

```
2022-05-18 22:37:44.092 22747-22747/com.yourcompany.podplay
E/xmlDateToDate: Unparseable date: "Thu, 10 Feb 2022 01:25 GMT"
    java.text.ParseException: Unparseable date: "Thu, 10 Feb 2022
```

```
01:25 GMT"
        at java.text.DateFormat.parse(DateFormat.java:362)
        at
com.yourcompany.podplay.util.DateUtils.xmlDateToDate(DateUtils.kt:5
9)
        at
com.yourcompany.podplay.repository.PodcastRepo.rssItemsToEpisodes(P
odcastRepo.kt:75)
        at
com.yourcompany.podplay.repository.PodcastRepo.rssResponseToPodcast
(PodcastRepo.kt:88)
        at
com.yourcompany.podplay.repository.PodcastRepo.getPodcast(PodcastRe
po.kt:61)
        at
com.yourcompany.podplay.repository.PodcastRepo$getPodcast$1.invokeS
uspend(Unknown Source:15)
        at
kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(Cont
inuationImpl.kt:33)
        at
kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:106)
        at android.os.Handler.handleCallback(Handler.java:883)
        at android.os.Handler.dispatchMessage(Handler.java:100)
        at android.os.Looper.loop(Looper.java:214)
        at
android.app.ActivityThread.main(ActivityThread.java:7356)
        at java.lang.reflect.Method.invoke(Native Method)
        at
com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(Runtime
Init.java:492)
        at
com.android.internal.os.ZygoteInit.main(ZygoteInit.java:930)
```

Notice that in the first line the tag now shows up as `E/xmlDateToDate` rather than `E/AndroidRuntime`. The stack trace still exists for reference, but the app successfully recovered from the thrown exception.

Great work on fixing this bug! :]

Wouldn't it be great, though, if crash data from real users came to you rather than manually searching through the code or trying to reproduce crashes in order to get the stack trace? Well, there's a way to do that. Next, you'll learn how **Firebase Crashlytics** can provide some automation around tracking crashes.

Firebase Crashlytics

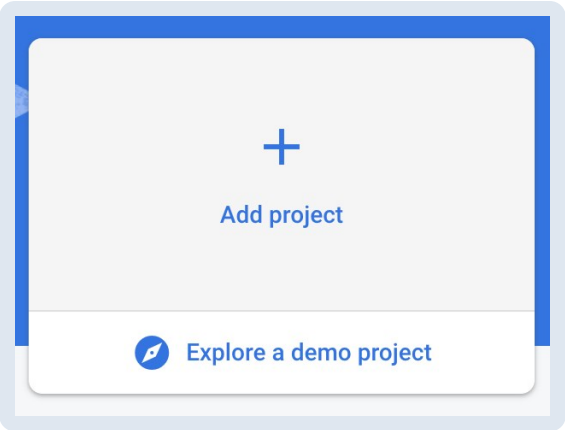
Crashlytics is an invaluable tool that provides you with crash reports, non-fatal errors, and Application Not Responding (ANR) errors. Firebase is Google's suggested app development platform which hosts Crashlytics as a service, along with several other great tools.

Next, you'll learn how to set up a Firebase account and connect it to the Podplay

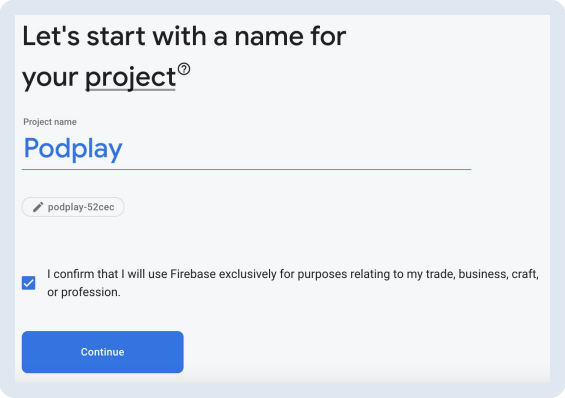
project. You'll then use Crashlytics to view crashes that occur in Podplay.

Creating a Firebase Project

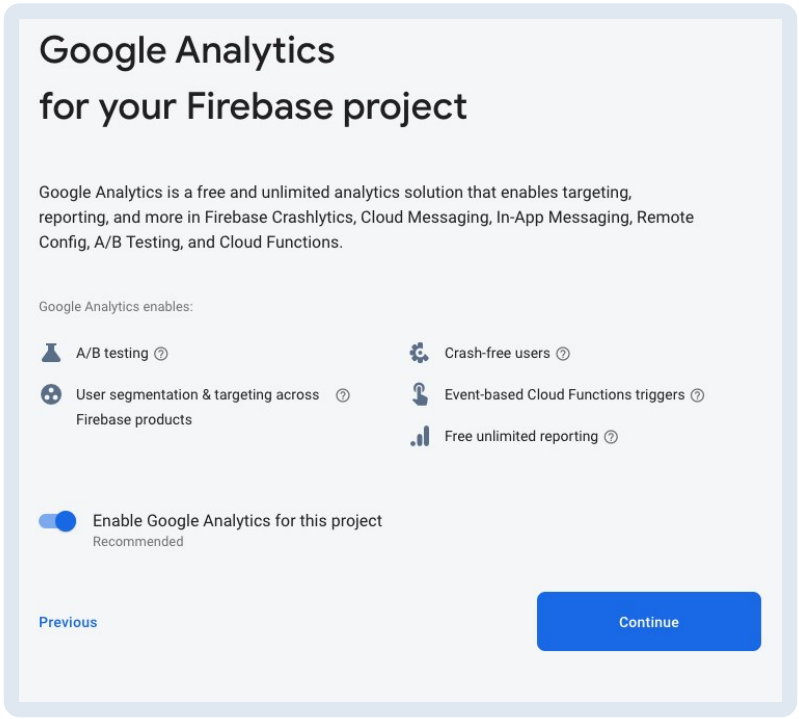
To start, go to the Firebase Console and set up a new Firebase project. Click **Add project**.



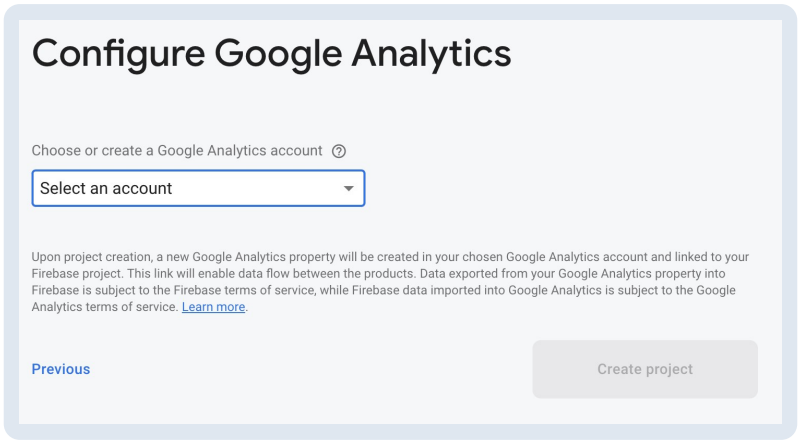
Next, enter the project name. It can be Podplay, similar to the app, or anything that'll help you correlate the Firebase project to your app. Check the confirmation checkbox and select **Continue**.



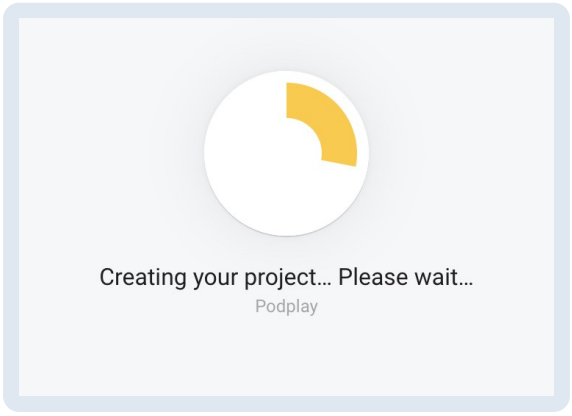
The next step will ask you to enable **Google Analytics** for the Firebase project. Crashlytics will use Google Analytics to determine crash-free user data. Make sure the toggle is on, and move forward using **Continue**.



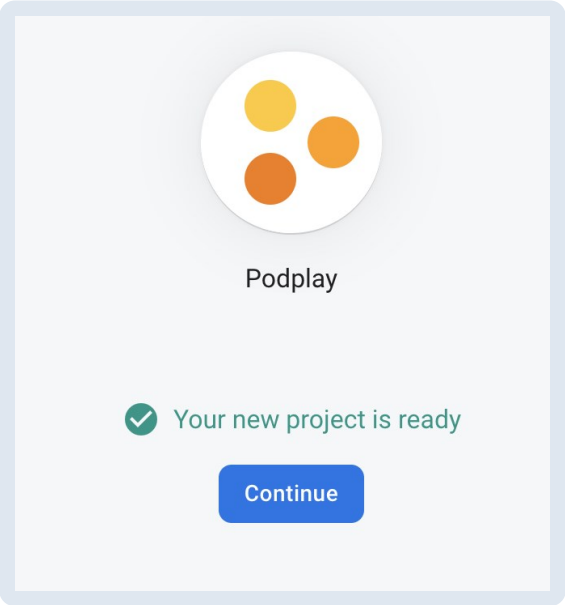
In the last step, choose the Google Analytics account to which you want the Firebase Project linked. Then click **Create project**.



The project gets created, and you should see a loading indicator while the project finalization wraps up.



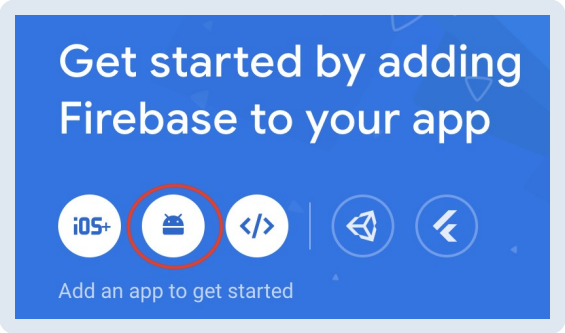
When the project is ready, select **Continue**.



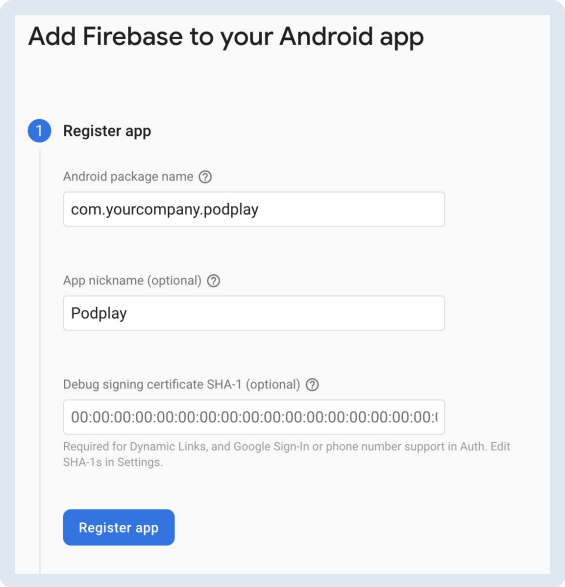
You'll redirect to the new Firebase project's dashboard. Now it's time to add your app to the Firebase project.

Adding Podplay to the Firebase Project

You can add the Podplay app to the Firebase project from the dashboard. Click the Android icon to start the process of adding Podplay.



The first step is to register your app. Enter in the package name of the Podplay app: **com.yourcompany.podplay**



You may also enter in an App's nickname if you'd like.

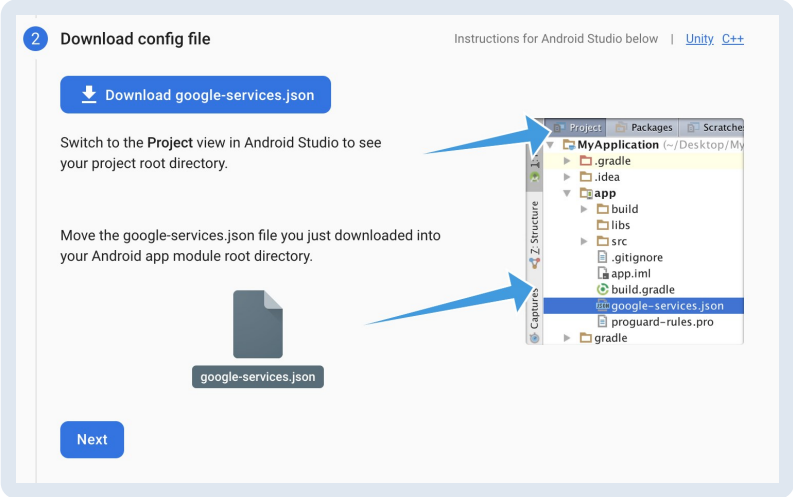
Also optional is the debug signing certificate SHA-1. This is useful for securing the Firebase project by only allowing app builds signed with the debug key on your workstation.

To get the SHA-1 of the debug certificate, open a **Terminal** window and navigate to the **home directory** on your workstation. Next, run the following command:

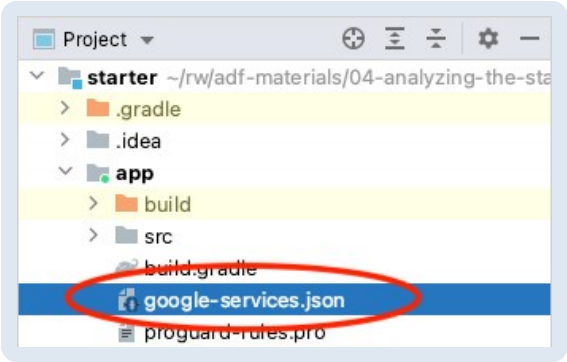
```
keytool -list -v \
-alias androiddebugkey -keystore .android/debug.keystore
```

If prompted for a password, enter **android**. Then, `keytool` will print out the certificate fingerprints for the `androiddebugkey` alias. Copy the **SHA-1** fingerprint and paste it in to the form field.

Next, click **Register app**. It'll prompt you to download the **google-services.json** file and add it to the **app/** folder. This file provides all the configuration data the app can use to communicate with the Firebase project.



Select **Download google-services.json**. After downloading, copy the file to the Podplay project's **app/** directory. In Android Studio, change to the Project view in the **Project** window, and you'll now see **google-services.json** in the directory.



Back on the Firebase console's app registration page, select **Next**. You'll need to update your project-level and app-level **build.gradle** to include the Firebase dependencies.

Open the project-level **build.gradle** first, and add the following class paths to the **dependencies** list:

```
classpath 'com.google.gms:google-services:4.3.10'
classpath 'com.google.firebase:firebase-crashlytics-gradle:2.9.0'
```

> **Note**: The Firebase setup also mentions the need to add the `google()` maven repository, but these have already been added to **build.gradle**.

Now, open the app-level **app/build.gradle**. Add the following plugins below the list of other applied plugins:

```
apply plugin: 'com.google.gms.google-services'
```

```
apply plugin: 'com.google.firebase.crashlytics'
```

Add the imports for the Firebase BoM, bill of materials platform, and specify the usage of the Firebase Crashlytics and Analytics libraries:

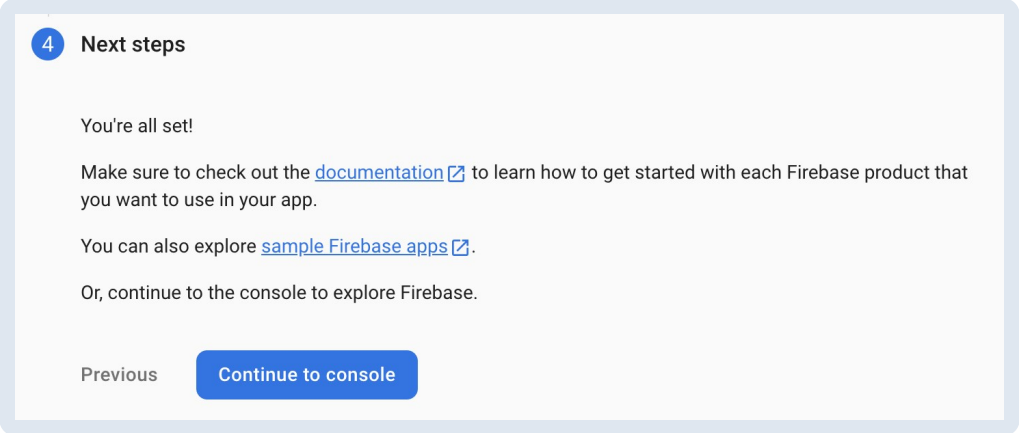
```
implementation platform('com.google.firebase:firebase-bom:30.0.1')
implementation 'com.google.firebase:firebase-crashlytics-ktx'
implementation 'com.google.firebase:firebase-analytics-ktx'
```

> **Note**: With the use of the Firebase BoM platform, you don't need to specify the versions on the individual libraries as the platform is aware of what versions to use.

Click the **Sync project with gradle files** icon in the toolbar to pull down the Firebase SDK into the project.

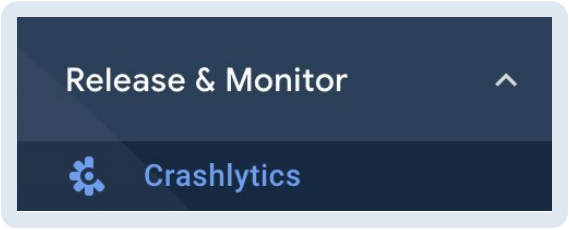
Once the gradle sync finishes, rerun the Podplay app.

Now, switch back to the Firebase web page and click **Next**. You're all done with the setup! Choose **Continue to console** to return to the Podplay Firebase project's dashboard.

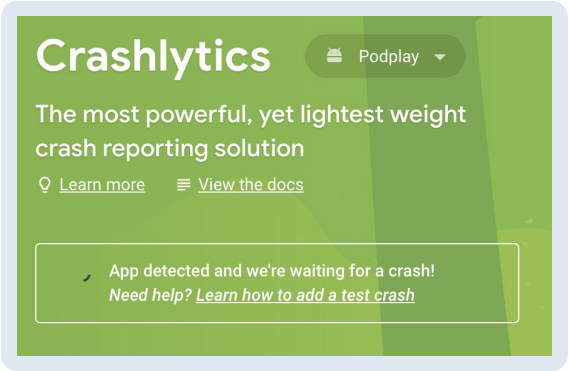


Previewing Crashes Within Crashlytics

From the Firebase project dashboard, select the **Crashlytics** option under **Release & Monitor** on the left-hand column.



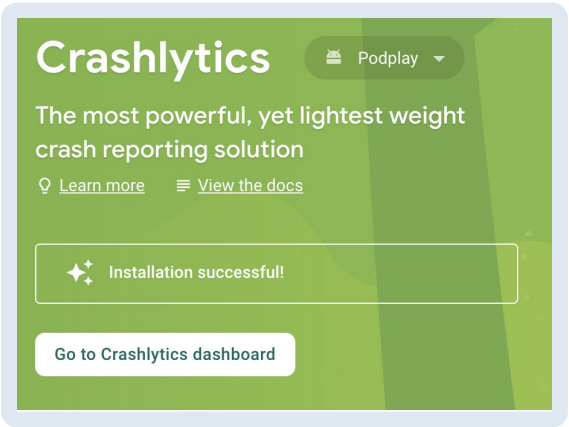
You'll see that the Crashlytics page is waiting for the app's first crash.



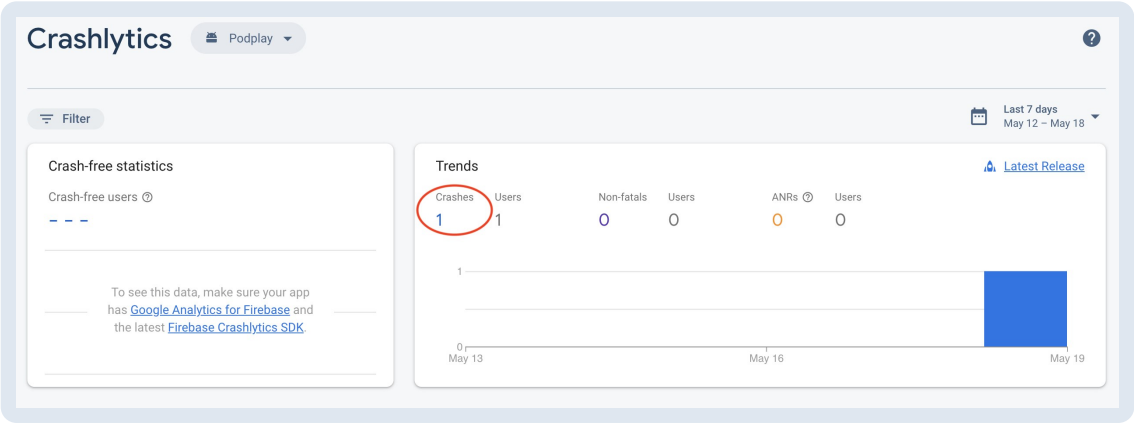
Now you need to crash the app. In Android Studio, open **PodcastActivity.kt** and add the following line in `onCreate()` right after the `super.onCreate()` call:

```
throw RuntimeException("Test for Crashlytics!")
```

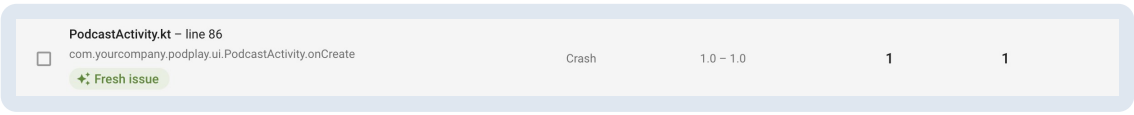
Run the app and go back to the Firebase project web page. You'll see that it has detected the crash and that the installation is complete! Click **Go to Crashlytics dashboard**.



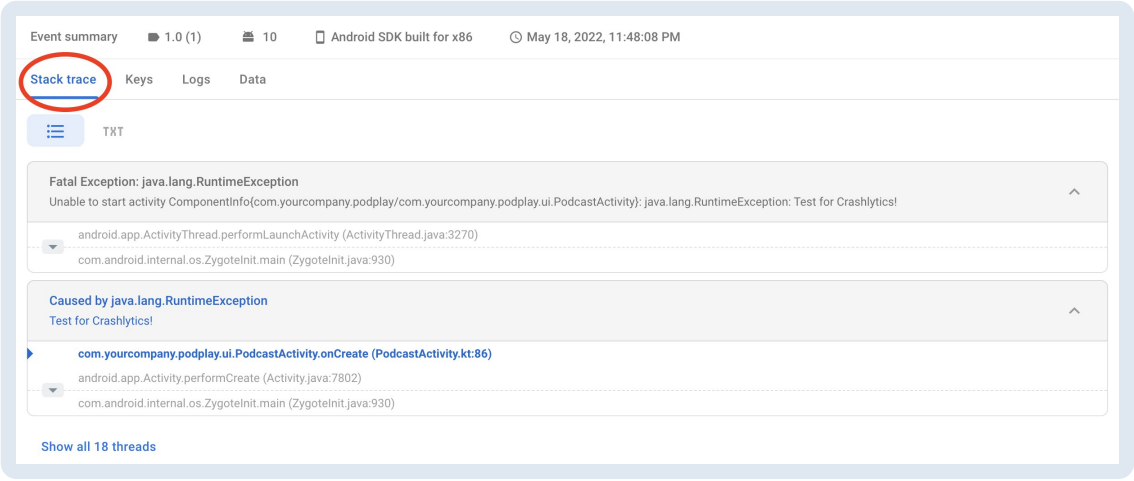
On the Crashlytics page, there's now one crash recorded:



Scrolling down a bit, you'll see the `PodcastActivity.kt` crash listed as a "fresh issue".

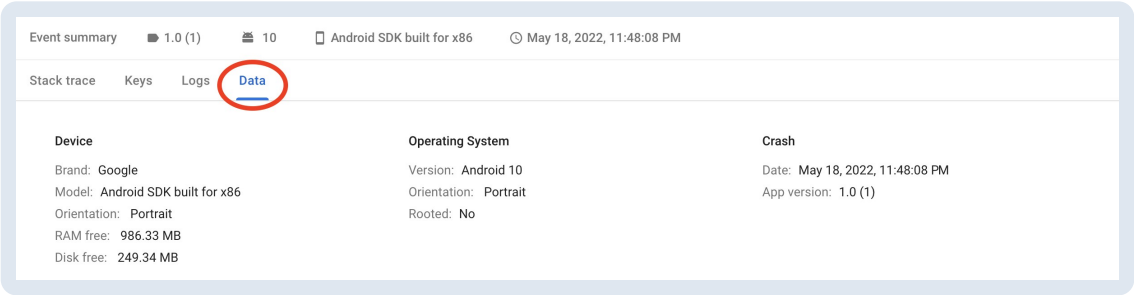


Click the issue to view the details page. Scroll down to the **Event summary** section to find the **Stack trace** tab with information on the stack trace itself.



You'll notice the `RuntimeException` with the test message. Click the down arrow to expand the stack trace fully.

The **Data** tab also provides more information about the crash, the device it occurred on and the OS running on it. This can be useful for pinpointing bugs specific to certain devices or operating systems or determining if there are any potential memory leaks.

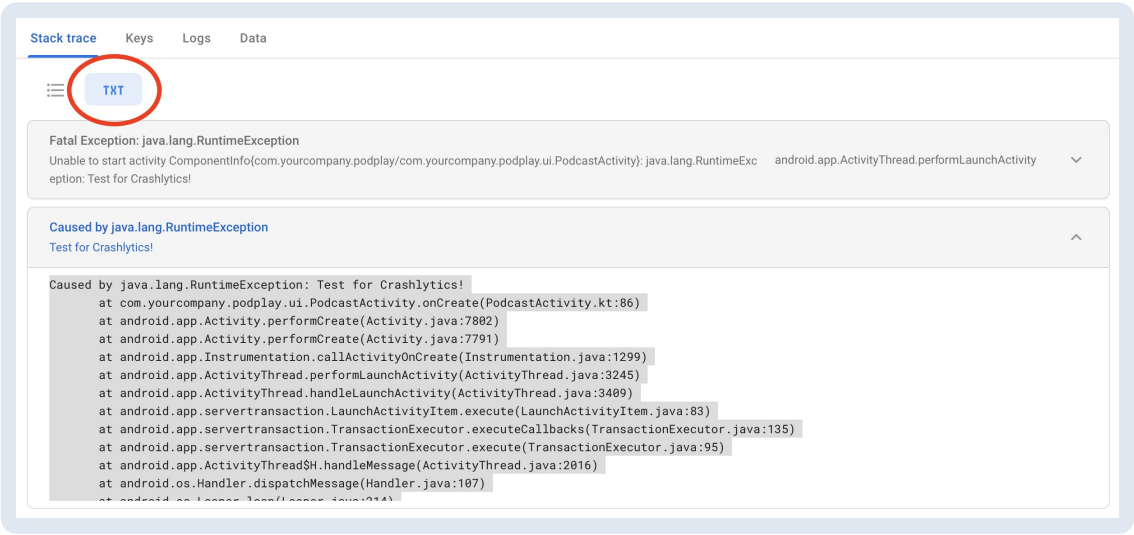


Great job on setting up Firebase Crashlytics! Now that you've got the crashes reporting to the Crashlytics service, it's time to try importing Crashlytics stack traces into Android Studio to inspect them.

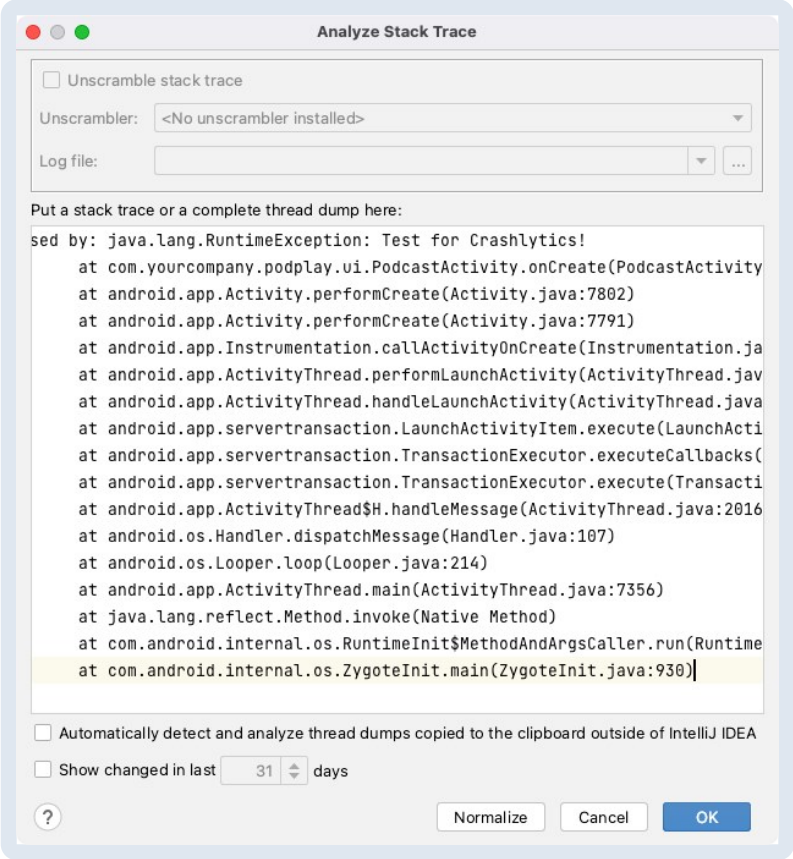
Analyzing External Stack Traces

Now that you've got a stack trace from Crashlytics, it's time to incorporate that with Android Studio to fix the bug you created!

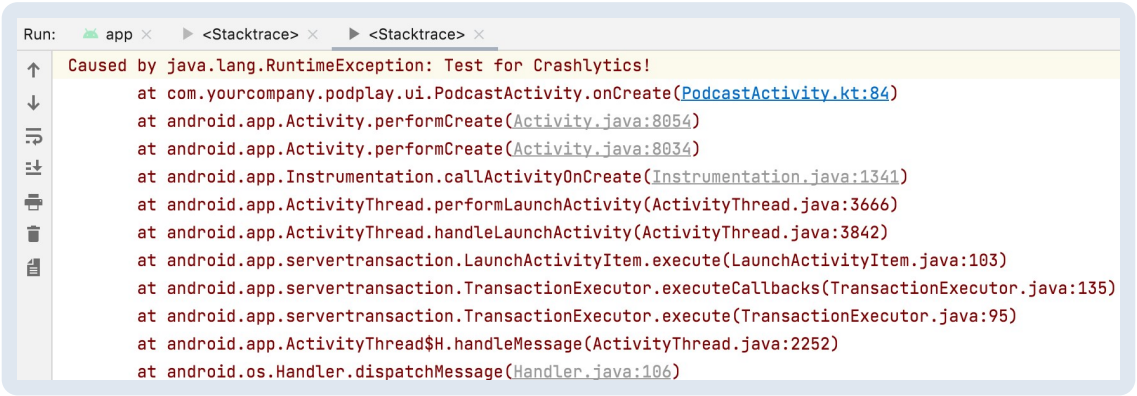
In the **Stack trace** tab, click the **TXT** tab to switch to the plain text view of the stack trace. Copy the stack trace in its entirety.



In Android Studio, navigate through toolbar options **Code ▸ Analyze Stack Trace or Thread Dump**. In the dialog window, paste the stack trace from Crashlytics and click **OK**.



A new tab in the **Run** window will appear with the stack trace provided. Text formatting will apply and the section **PodcastActivity.kt:84** will highlight as a link.



Click the link, and it'll take you to the line with your `throw RuntimeException()` call.

Delete that throw call, and you've fixed the app.

Congrats! You've now mastered the art of reading and analyzing stack traces.

Challenge

It's time to put what you've learned to use. There's another bug that is present in Podplay. Launch the app and search for any podcast. When the list of

podcasts displays, repeatedly tap one until the app crashes.

Use the skills you've learned in this chapter to find the stack trace and the underlying bug, and fix it.

Key Points

- A stack trace shows a list of methods called at a certain place in your code.
- Understand the method calls that led up to the crash by reading a stack trace.
- You can capture crashes and rethrow them as non-fatal errors.
- Use Firebase console to connect Crashlytics to any Android app.
- You can utilize Crashlytics to monitor crashes and find stack traces.

Where to Go From Here?

Firebase Crashlytics is a powerful tool that you have at your disposal. You can find more in-depth details in the Firebase Crashlytics documentation about the Crashlytics service and what you can do with it.

In the next chapter, you'll learn how to manipulate data at runtime while debugging your app. This will allow you to simulate certain situations and put your app in a state that replicates crashes that your users will experience.

5 Watches & Evaluating the Code

Written by Vincenzo Guzzi

A major part of debugging is interacting with and changing variables when your code is in a suspended state. This allows you to manipulate responses to simulate different scenarios and see how your app will react. The **Variables** pane inside the **Debug window** allows you to perform this manipulation of variables by providing useful tools, which you'll be exploring in this chapter.

You'll learn how to:

- Read and use the **Variables pane**.
- Make important variables more visible by using flagged and the **Watched pane**.
- Evaluate code at runtime.
- Change the values of variables to find bugs.
- Augment code with **Force Return** and **Throw Exception**.

Exploring The Variables Pane

The Variables pane, like everything else in the Debug window, is only viewable when your code is suspended. This pane contains all of the information about the variables that your current context can view. Within this pane, you can also manipulate the data by setting values, finding usages, or pinning important information.

To start, open the **Podplay** starter project in Android Studio. Open **EpisodeListAdapter.kt**, and add a breakpoint on the first line inside `onBindViewHolder()` so you can suspend your code inside this method.

Run the app in **debug mode**. Once PodPlay has loaded, tap the search icon, enter the term "All about android" and open the **All About Android (Audio)** podcast.

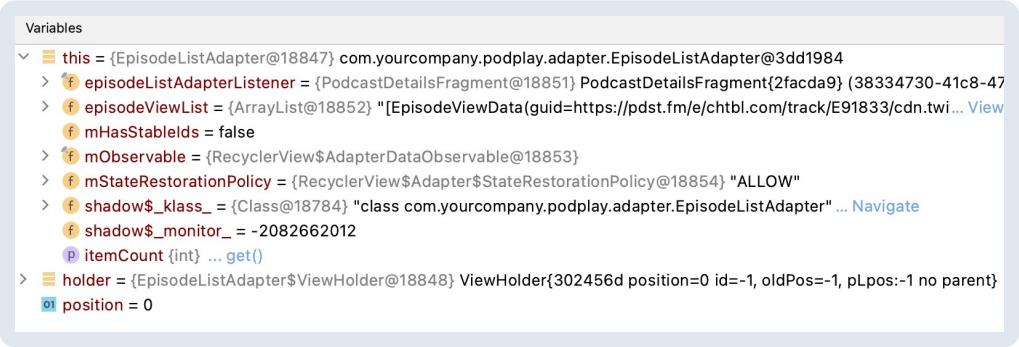
Your program is now suspended inside `onBindViewHolder()` where you placed your breakpoint. The Debug window automatically opens when your code suspends. If it doesn't, open it manually by clicking **View ▸ Tool Windows ▸ Debug**.

You'll locate the Variables pane to the right of the Debug window:

Each line inside the Variables pane is a variable; these can be primitives or objects. The first variable you see is `this`.

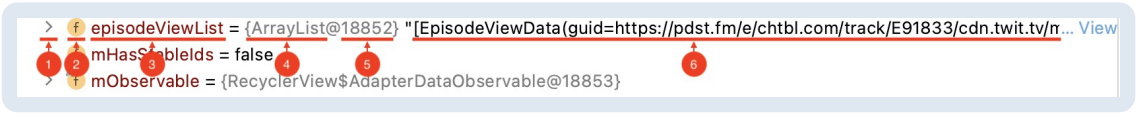
In Android, **this** refers to the scoped context. In this case, it's the context of `EpisodeListAdapter`. The Variables pane will always show the scoped context as the first variable in the list. That way you can find out what context is accessible from wherever your code is suspended.

There's a chevron located to the left of `this`, which means that it's an **object** containing additional values. Click the chevron to expand `this` and see its values.



Each variable in the pane provides the variable's name and its value. If it's an object, it will display metadata to help you understand what type of object it is.

In the second row of `this` is a variable called `episodeViewList`. This is what the row shows:



1. A chevron that you can click to expand the object and see the variables inside.
2. The variable identifier — this one indicates that `episodeViewList` is a **final field**.
3. The variable name.
4. The variable type.
5. The object ID.
6. The variable value.

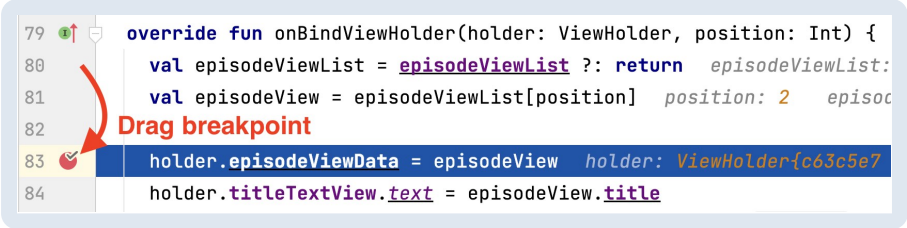
In this case, the local scoped context (`this`) will always appear in the first row of the Variables pane. The following variables in the list will be the ones that are either method parameters or ones created within the current scope.

The following two variables in the list are `holder` and `position`. Both of them have been passed into `onBindViewHolder()` as parameters.

Click **Step over** twice to execute the next two lines of code.

You'll now see that two new variables have appeared in the Variables pane; `episodeViewList` and `episodeView`. These local variables have *just* been created in the two lines you just executed.

Drag and drop your breakpoint to your new line, so future suspends will occur on this line:

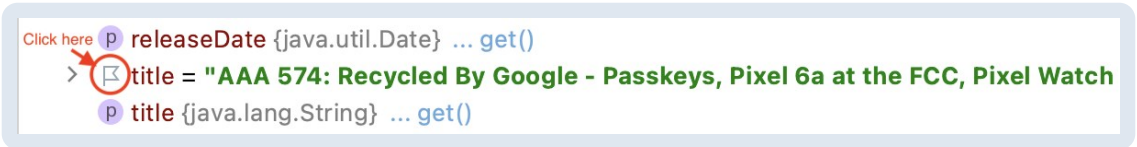


Pinning and Watching Variables

The Variables pane is often full of information. The pane itself doesn't know what variables are most important to you. However, it does provide a way for you to let it know.

The first and easiest way to make variables that you care about more visible is to pin them to the top of an object. For now, Android Studio only allows you to pin variables that are within an object and not in the top level of the Variables pane.

Do this for the `String` variable `title`. Find it within `episodeView` by clicking the chevron to expand `episodeView`. Then scroll down and tap the **variable type** identifier of `title`. It will turn into a flag icon when your mouse hovers over it.



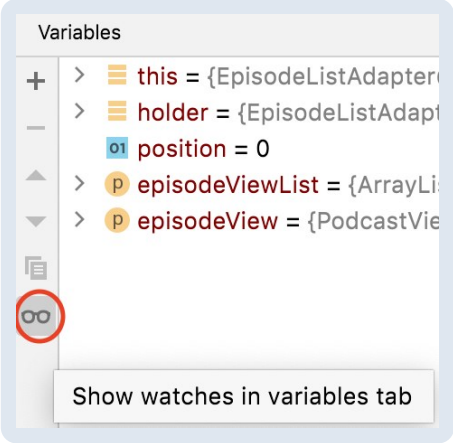
The first item within `episodeView` will now be `title` and it will stay there in subsequent debugging sessions.

You've now also changed the variable type icon to a *blue flag* to indicate that it

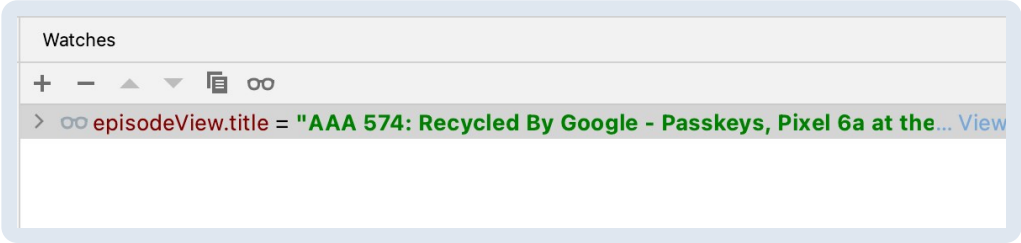
has been pinned. To unpin a variable, simply click the flag icon again, and it will go back to its ordered position.

For super important variables that you want to be able to monitor all the time, you can add a variable to your **Watches pane**.

To show your **Watches** pane, click the glasses icon to the left of the Variables pane:



Add `title` to your Watches by selecting the + icon inside the **Watches pane**. Type in `episodeView.title` and then press **Return**:



`episodeView.title` will now be permanently shown inside your **Watches** pane. Click **Resume Program** twice to see your watched variable update each time.

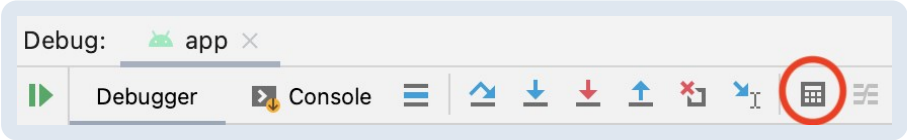
Use the + icon to add another watch for `episodeView.duration`:

Simply right-click a watched variable and select **Remove Watch** to remove a watch.

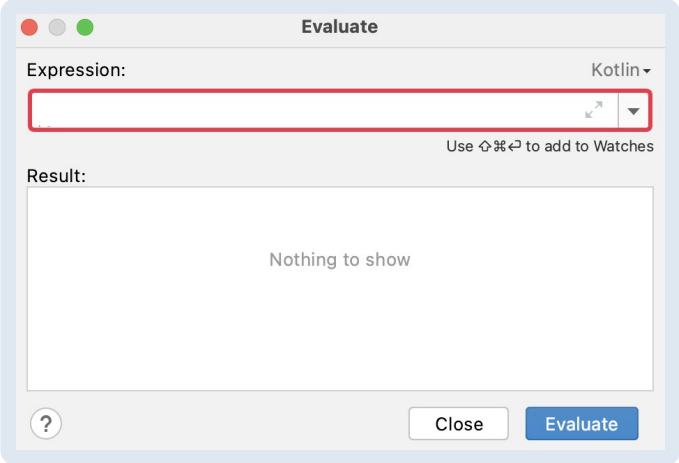
Evaluating Code

Whenever your code is in a suspended state, you have access to retrieve and manipulate everything available to your current scope. The most instantaneous way to manipulate your code is with **Evaluate Expression**.

You'll find Evaluate Expression at the end of the debugging actions toolbar:



Click **Evaluate Expression** to open up the **Evaluate** window:



Here, you can enter any code you like, and if your scoped suspended position can evaluate the code, you can do that here too.

You can use the **Evaluate** button when debugging for many use cases; see below for a few examples that you can try yourself in the current scope:

- Checking over data content by running:

```
episodeView.description.contains("Android")
```

- Seeing if a primitive type is greater than a certain value:

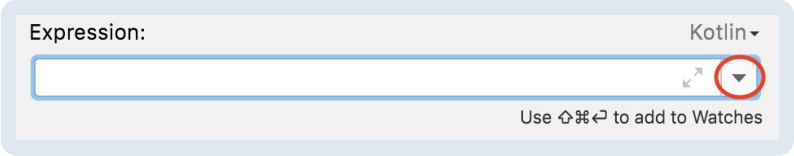
```
position > 10
```

- Running logic that hasn't yet been executed:

```
DateUtils.dateToShortDate(episodeView.releaseDate!!)
```

If you want to see all of your previously evaluated expressions, use the small

arrow to the right of the Expression window.



The ability to evaluate code is a powerful tool when debugging. As well as debugging your existing logic and data, you can use evaluating code to test logic that you haven't yet coded so. Then you can see if it works as intended before committing it to a real method. As your code doesn't have to be re-compiled to test different logic, the speed at which you can try different code saves a lot of time!

Augmenting Code

Code augmentation is the act of changing the value of a variable to something different from its real value. You can use this to simulate different data responses and evaluate how your app UI responds.

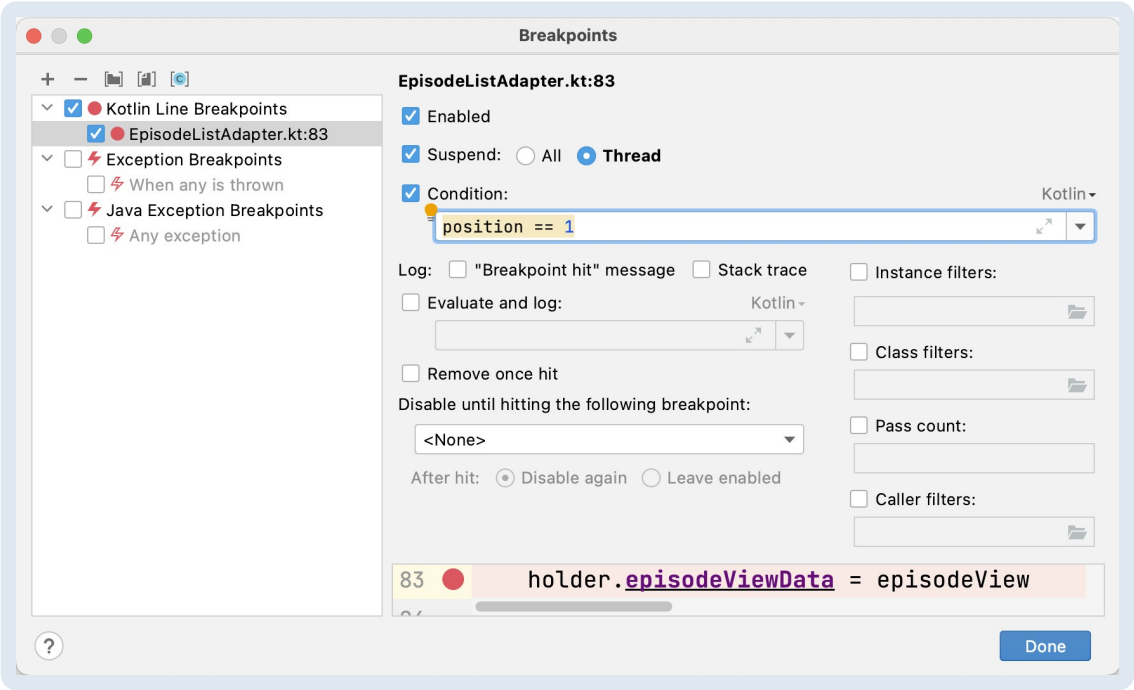
There are a few different tools within Android Studio that you can utilize to augment code. The easiest way to do this is to directly change the value of a variable when your program is suspended.

If your program is running, stop it by clicking the **Stop** icon.

Right-click the breakpoint inside `onBindViewHolder()` and click **More** to expand the **Breakpoints window**.

Check the **Condition** box and enter the condition of:

```
position == 1
```



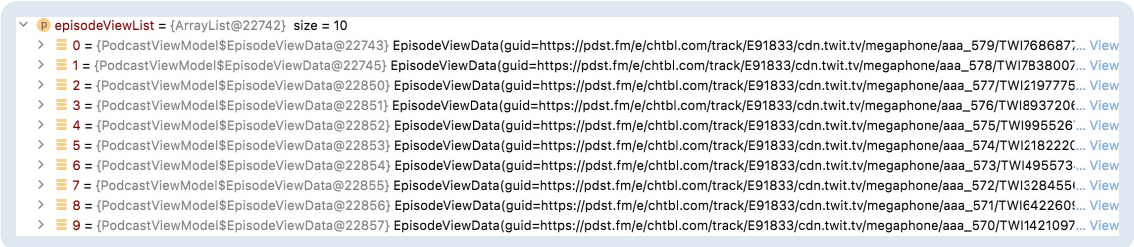
Click **Done**.

Your breakpoint will now only suspend the first time `onBindViewHolder()` is called; this is when you'll augment your code.

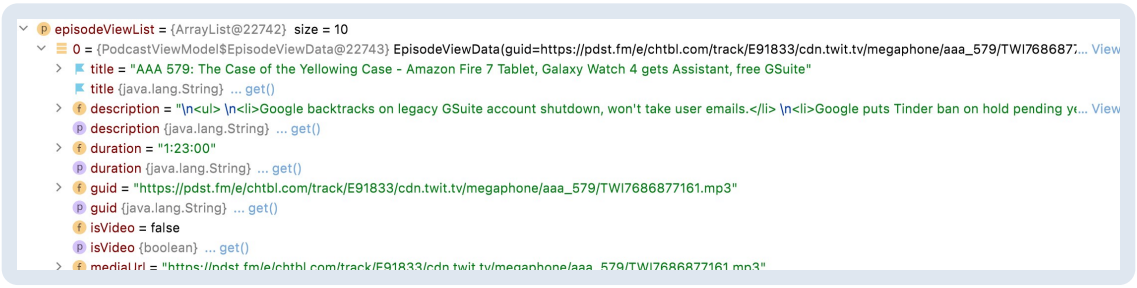
Build and run your app again in debug mode. Search for and click **All About Android (Audio)** again, so your app suspends on your breakpoint.

You're going to debug how your program behaves when there's an unexpected empty `String` in the variable `mediaUrl`.

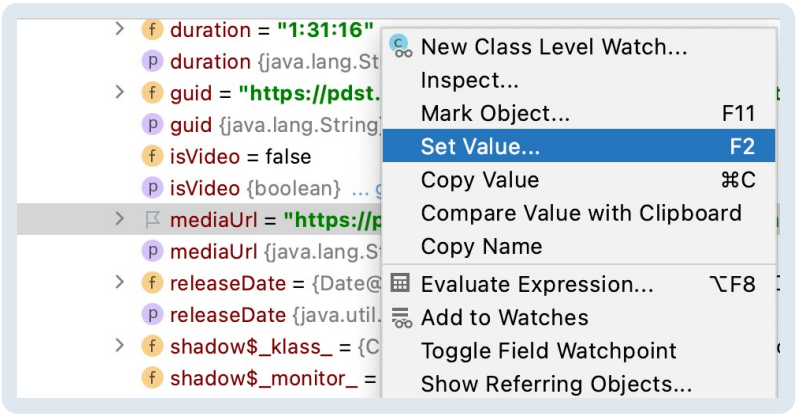
In the **Variables** pane, expand `episodeViewList`:



You're now viewing all of the podcast episodes in an `ArrayList`. Expand the first item by clicking the chevron next to **0**:



This is the podcast episode that you're going to augment. Scroll down until you see the variable `mediaUrl`, now right-click the variable and select **Set Value...**:



Replace the `mediaUrl` value with an empty text and press **Return**:

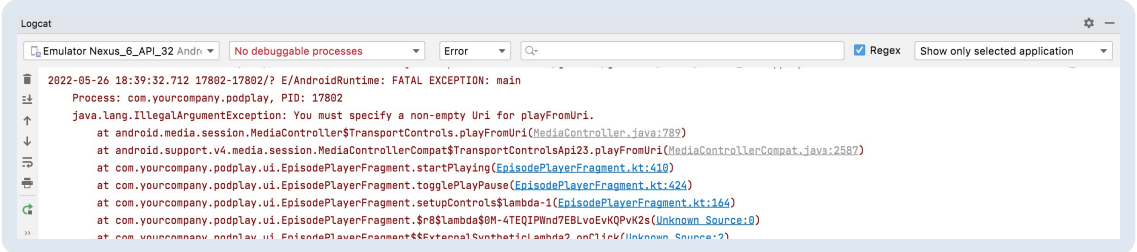
Now, click the **Resume Program** icon so that you can see how your program responds.

With PodPlay running, select the first podcast episode in the list, the one that you augmented.

Now try and play the podcast episode by clicking the **Play icon**.

Ouch! Your app crashed.

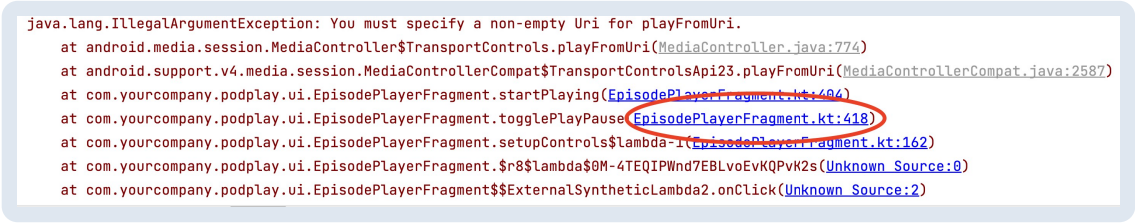
Open the Logcat window, and you'll see an **IllegalArgumentException** stating: "You must specify a non-empty Uri for playFromUri."



This is bad as `mediaUrl` is a variable that an external API sends. You can't

expect it always to be present. Your logic needs to handle edge cases where data is empty like this.

Click the crash link for `togglePlayPause` to go to the method that threw the `IllegalArgumentException`.



You can use **Evaluate expression** to quickly test a fix for this crash.

Remove the previous breakpoint and place a new one on the first line inside `togglePlayPause()`.

Now build and run your app again in debug mode and go through all of the previous debugging steps:

1. Search for “Android”.
2. Open the **All About Android (Audio)** podcast.
3. Change the value of `mediaUrl` within the first element of `episodeViewList` to an empty `String`.
4. Resume your program by clicking the **Resume** icon.
5. Tap on the first podcast episode on the list.
6. Play the podcast episode by tapping the **Play** icon.

Your app should now be suspended inside `togglePlayPause()` right before the crash occurs.

Click **Evaluate Expression** so you can see which code to use to capture this scenario.

Try to find the correct code to use for checking if the `mediaUrl` text is empty by using **Evaluate Expression**.

Did you find the answer?

If not, here is the winning logic:

```
podcastViewModel.activeEpisodeViewData?.mediaUrl?.isEmpty()
```

You just saved yourself a lot of time running through the debug steps multiple

times to test different pieces of logic until you found the correct one.

Now you can add the `isEmpty` check at the first line of `togglePlayPause()`:

```
if (podcastViewModel.activeEpisodeViewData?.mediaUrl?.isEmpty() ==
true) {
  Log.d("test", "MediaURL is empty, unable to play podcast
episode.")
  Toast.makeText(context, "Unable to play podcast episode, media
URL missing.", Toast.LENGTH_SHORT)
  return
}
```

Make sure you add the **Log** and **Toast** imports to the top of **EpisodePlayerFragment.kt**:

```
import android.util.Log
import android.widget.Toast
```

This code will now check if the `mediaUrl` is empty. If it's empty, it'll display a toast to the user and return from the method instead of crashing.

Go through the previous debug steps again to try out your new fix if you'd like!

Force Return

Changing the values of variables during code suspension is a powerful debugging tool, but what if you wanted to augment the returning value of a method? You can do this with **Force Return**.

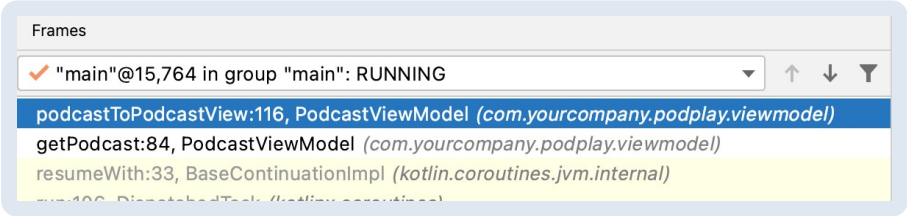
Force Return allows you to end your current scope and return an evaluated expression to the scope above it. Like direct code augmentation, it's a great way to test different scenarios of what a method might return.

To demonstrate **Force Return**, remove all of your existing breakpoints by selecting **Run ▸ View Breakpoints...** and deleting them.

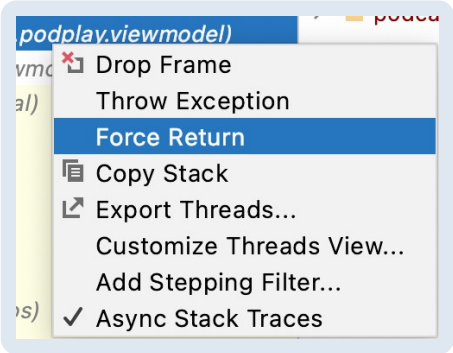
Then, add a breakpoint onto the first line inside `podcastToPodcastView()` located in **PodcastViewModel.kt**:

Next, you'll see how your UI behaves if a podcast doesn't have a title or description in this scenario.

Build and run your app again in debug mode, search for a podcast and select the first one. **Force Return** lives in the **Frames** pane that you explored in a previous chapter. In the **Frames pane**, you can see that the currently selected frame is the method that you're suspended in; `podcastToPodcastView()`:

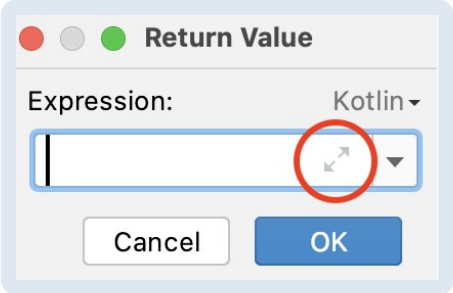


Right-click your frame and select the option **Force Return**:



A new window will now be visible called **Return Value** with an expression input box. This is the place where you'll return a new, augmented data value.

Select the **expand** icon next to the expression box, so you have more visibility of the code you're inputting:



Now, return a `PodcastViewData` object that has an empty title and description:

```
PodcastViewData(podcast.id != null, "", podcast.feedUrl, "",
podcast.imageUrl, episodesToEpisodesView(podcast.episodes))
```

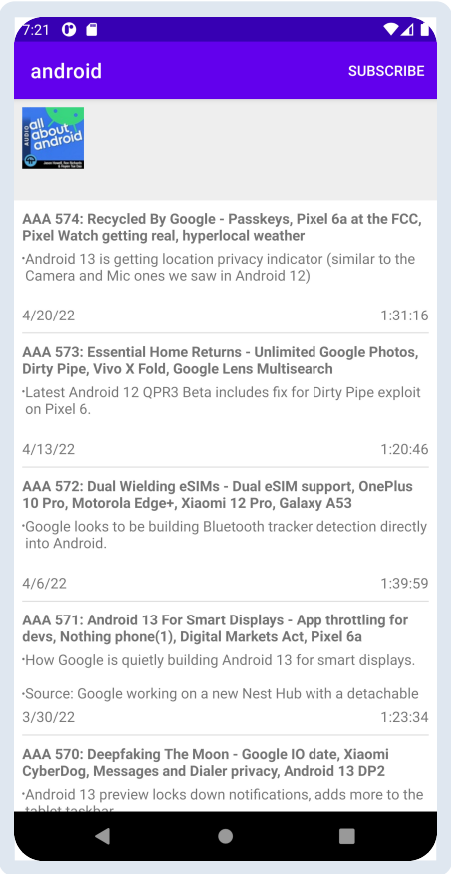
This was copied from the existing method logic within `podcastToPodcastView()` with the title and description parameters replaced.

Click the **collapse** icon and then click **OK**.

> **Note**: Android Studio 2021.2.1 has an issue when pressing **OK**; the Return Value window doesn't close automatically. Clicking **OK** again will result in an error that states "Error while doing early return: Thread has been resumed." To overcome this issue, press **Cancel** after pressing **OK** the first time.

Your frame will now have been returned, but your app is still suspended; resume your app by clicking the **Resume Program** icon.

You have now successfully augmented your code with **Force Return**, you'll now see your podcast without a title and description within your UI. You're also free to leave it as it is or change how your app behaves when this data is missing.



Throwing Exceptions

You can also augment code by throwing exceptions. Throwing rogue exceptions at your program is a great way to see how resilient your code is.

Remove your existing breakpoint in `PodcastViewModel`. Then open **PodplayMediaCallback.kt** and scroll down to `setState()`.

Within `setState()` you'll see that halfway through the method, there's a try-catch statement that wraps around this code:

```
mediaPlayer.playbackParams =
mediaPlayer.playbackParams.setSpeed(speed)
```

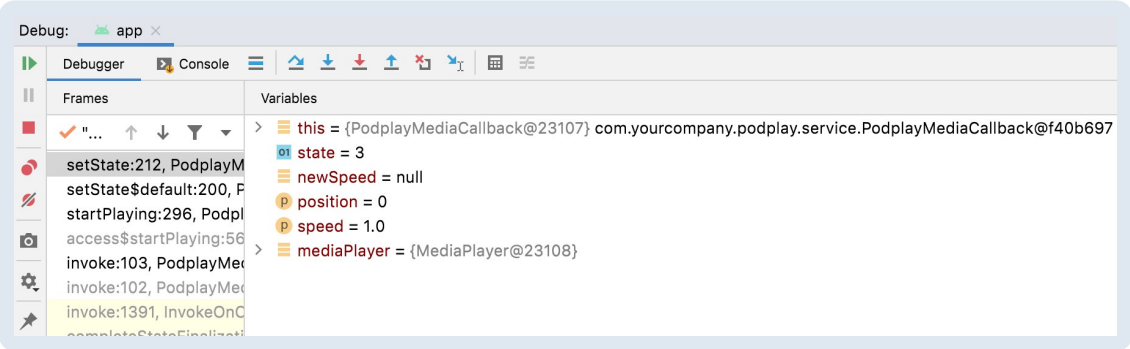
If `setSpeed()` throws an exception, a set of logic is run. The only trouble is that it's hard to verify that the exception logic works if `setSpeed()` always works. This is where you can augment the code to throw an exception.

Remove the last added breakpoint and set a new breakpoint on the line from the previous code block:

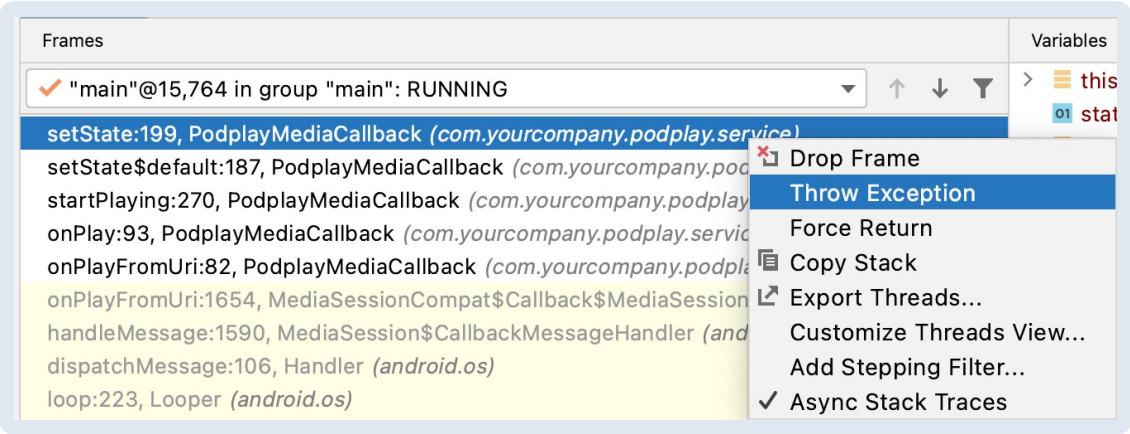
```
198     try {
199       mediaPlayer.playbackParams = mediaPlayer.playbackParams.setSpeed(speed)
200     } catch (e: Exception) {
```

Now, build and run PodPlay in debug mode.

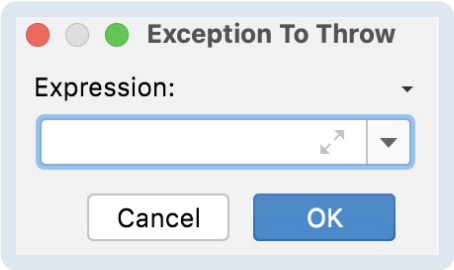
Suspend the code on this breakpoint by searching for a podcast, selecting an episode, and then playing that podcast episode.



Your code is now suspended on `setSpeed()`. Normally, this method would execute, and your catch statement wouldn't be hit. For this case, right-click your frame inside the **Frames** pane and select **Throw Exception**.



You'll see the new **Exception To Throw** window:



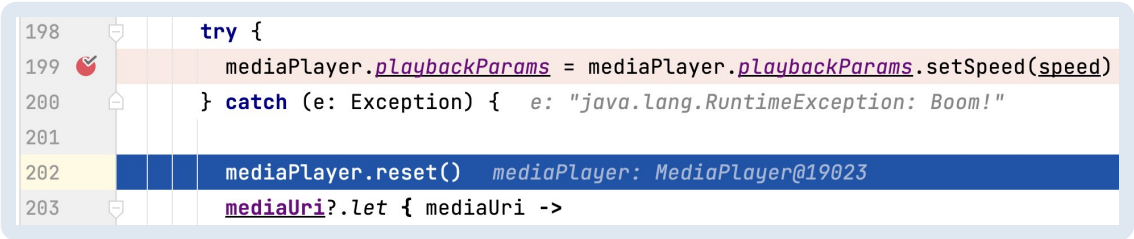
Enter the exception that you'll force return:

```
RuntimeException("Boom!")
```

Click **OK**.

Your code has now landed within the catch statement!

Click **Step over** once to move down into your `catch` scope, and you'll see the exception that you just created, with the message "Boom!" has been caught.



Click **Resume Program**, and your code will suspend again on the `setSpeed()` method, which is the correct functionality in this instance, you can rest assured knowing that your catch statement is behaving correctly.

Key Points

- Use the Variables pane to observe your local and context-accessed variables.
- Use Variable pinning and the Watches pane to keep important variables in view.
- You can use Evaluate Expression to debug your code logic without re-running your app.
- Use Force Return and Throw Exception to debug how your app behaves in unusual scenarios.

Where to Go From Here?

You've learned how to observe and change variables within Android Studio to create better, more resilient code, and you've learned how to use the Variables

pane to help you find and fix problems, but there's still more to learn!

Here are some techniques you might be interested in:

- How to interpret the different colors of inspected variables.
- Viewing variables in-line and within tooltips.
- Comparing variable values with your clipboard.

Check out the JetBrains documentation for examining a suspended program to learn about these techniques in-depth and much more.

6 Layout Inspector

Written by Zac Lippard

Up until now, you've learned many ways to debug your app's code. But what if you wanted to debug the UI? Enter, the **Layout Inspector**.

The Layout Inspector is a tool provided by Android Studio that enables you to debug layouts in your app. The inspector takes a snapshot of the UI to provide details about the layout. These details can be what views are in the UI hierarchy, attributes of each view and a 3D rendering to inspect the views at different angles.

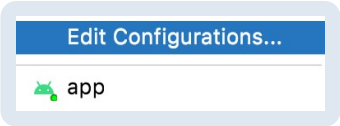
Effective use of the Layout Inspector can help you catch potential defects in your layouts, as well as help improve the overall UI performance.

In this chapter, you'll use the Layout Inspector and learn how to:

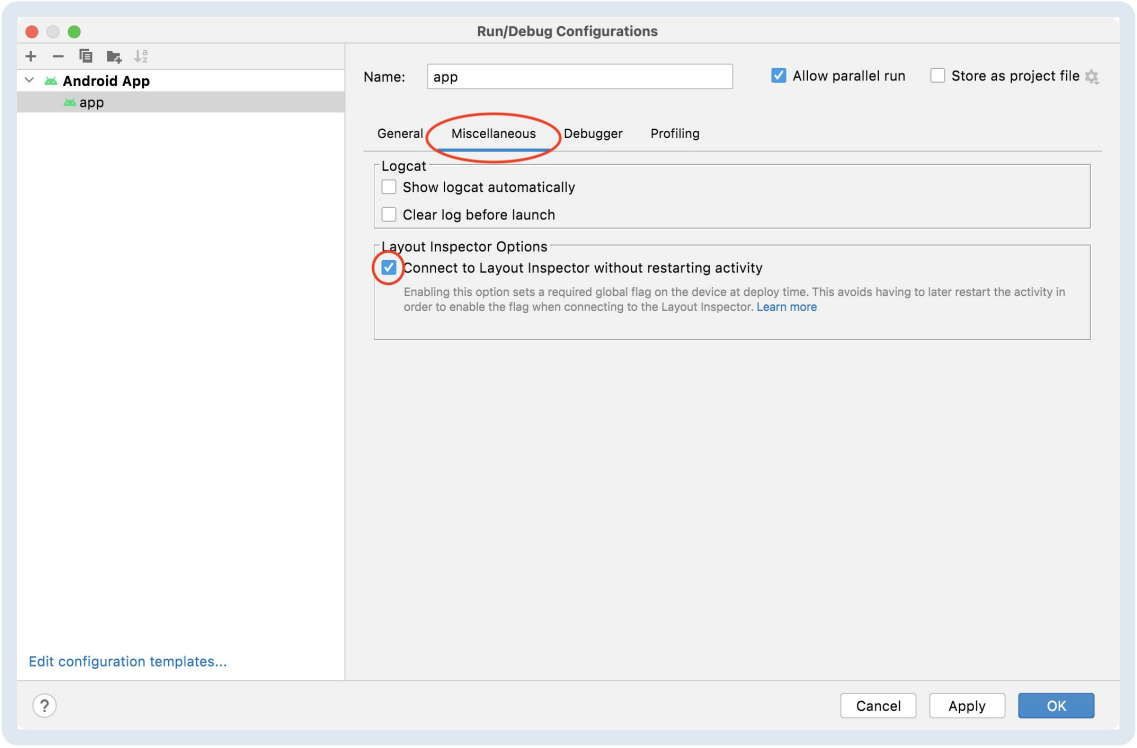
- Inspect the view hierarchy.
- Generate a 3D rendering of your app's UI.
- Read the attributes of a view.
- Condense layout layers.
- Validate layouts on different devices and configurations.

Running the Layout Inspector

Begin by opening the **Podplay** starter project in Android Studio. Before running the app, click **Select Run/Debug Configuration** from the toolbar and choose **Edit Configurations...**



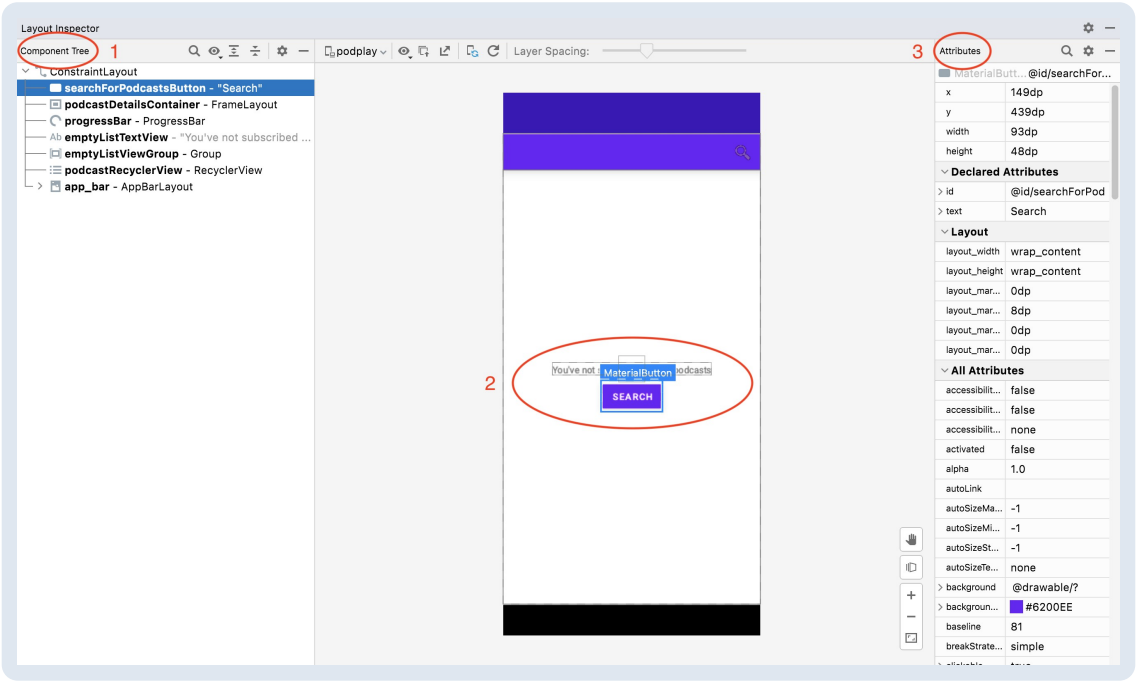
In the **Run/Debug Configurations** dialog, click the **Miscellaneous** tab and check the **Connect to Layout Inspector without restarting activity** box.



When you check this option, the Layout Inspector passes a flag to the launching activity. The flag allows the inspector to connect to the app immediately. Otherwise, the inspector has to restart the activity each time it's loaded, and you'll have to navigate back to the view you want to inspect.

Click **OK** to save changes.

Run the app. When the app launches, start the Layout Inspector by clicking **Tools ▸ Layout Inspector** from the Android Studio menu.



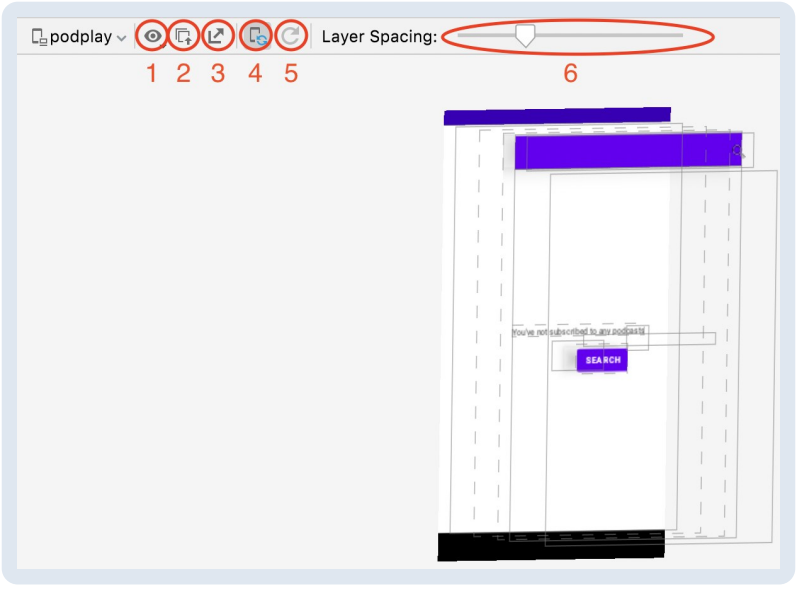
The following sections in the inspector include:

1. **Component Tree**: This shows all the views in the hierarchy. You'll notice this particular layout's hierarchy is shallow, with only two layers.
2. **Layout Display**: This is the rendered display of your layout. This section allows you to zoom in on specific views in the layout and explore the layers of the hierarchy in 3D.
3. **View Attributes**: When you select a view from the component tree or the layout display, the attributes for that view appear here.

At the bottom right of the layout display section click the **3D Mode** button.

> **Note**: The first time you do this, it may ask you to install some additional components within Android Studio.

The display rendering updates to show a 3D layering of your views:



Notice there's a toolbar at the top of the layout display section. This toolbar includes several features such as:

1. Showing and hiding certain aspects of views.
2. Loading overlays.
3. Exporting snapshots.

4. Enabling live updates of the inspector.
5. Refreshing the inspector.
6. Adjusting the spacing of the hierarchy layers.

You'll learn more about each of these later in this chapter.

The 3D rendering in the layout display allows you to observe the layout and its layers, view outlines display and show you how many nested view groups are in your layout. This rendering can help you determine if you have too many layers and if your layouts are complex.

When the UI gets rendered, the system has to do a number of layout passes to draw each view layer in the hierarchy. Too many nested layers can cause drawing issues like **overdraw** and will slow down the layout process.

Overdraw occurs when the system attempts to draw over the same pixel or group of pixels in layout passes. Overdraw is inefficient and often unnecessary as the GPU has to re-process rendering and draw the same part of the screen multiple times.

> **Note**: To learn more about overdraw and ways to reduce it, read the Overdraw Android developer documentation.

The layout display will show you potential overdraw areas within the 3D rendering. Overlapping layers with background colors set in the same region are candidates for overdraw. You should avoid this as much as possible.

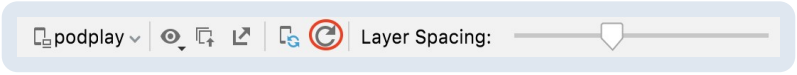
Now that you've understood what the Layout Inspector can do, it's time to improve one layout in the Podplay app!

In the starter project, **fragment_episode_player.xml** contains a number of nested `RelativeLayout` and `LinearLayout` view groups. You'll improve this layout file and use the Layout Inspector to view the changes.

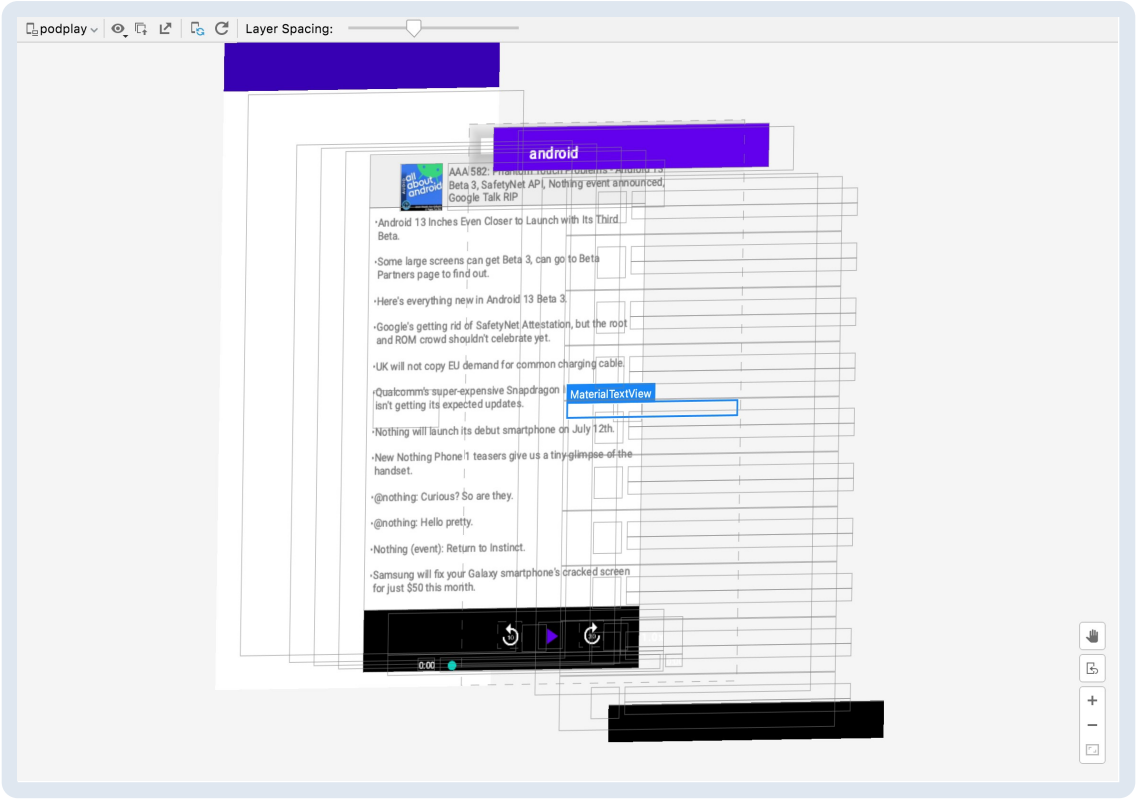
Refreshing the Layout Inspector

You'll need to refresh the inspector to load the episode player view. To do so, go to the running Podplay app, tap **Search** and enter "android". Tap a podcast from the list and choose an episode so the player screen loads.

Go back to the Layout Inspector and click the **Refresh layout** icon.

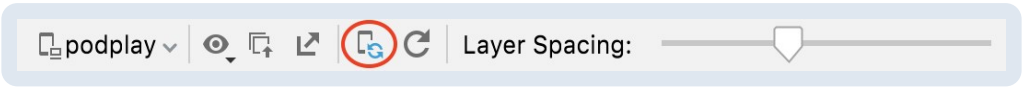


The layout display will refresh and show the hierarchy for the episode player fragment.



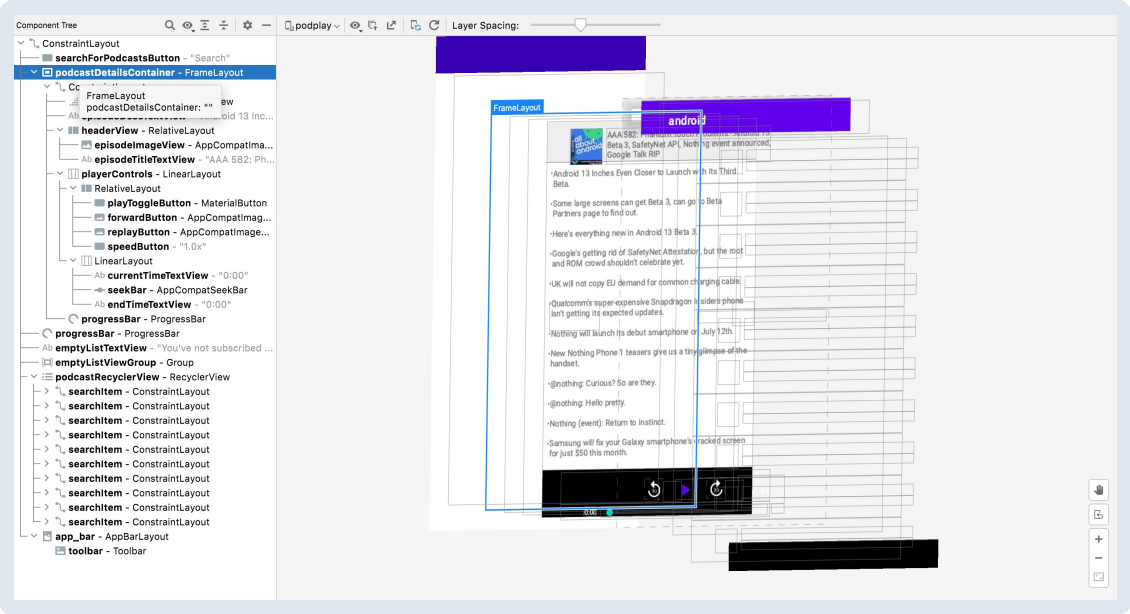
As an alternative to manually refreshing the Layout Inspector, you can enable the inspector to update automatically. Whenever you go to a new screen or change the UI, the inspector will update its component tree, display and its view attributes accordingly to reflect the changes from the app.

Click the **Live Updates** icon in the inspector toolbar to use automatic updates.



Isolating a View

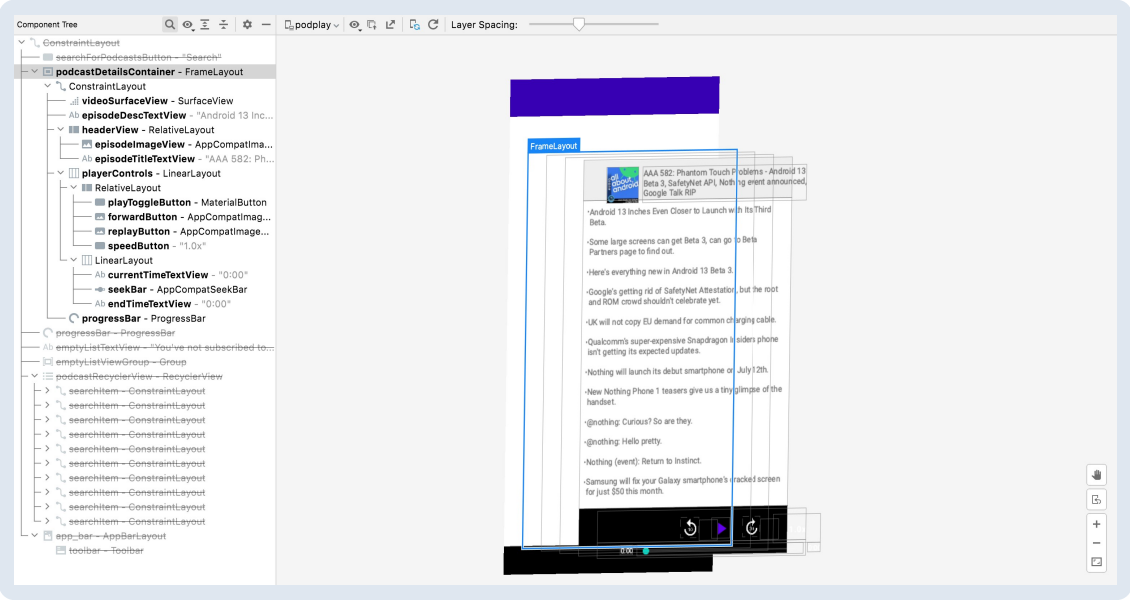
There's a lot going on in the current layout display with all the views in the hierarchy. What you'll focus on is the **podcastDetailsContainer** view. Click that view in the component tree, highlighting the associated view in the display.



This is a bit better, but you can also isolate this container view and remove all the others in the hierarchy that you don't care about.

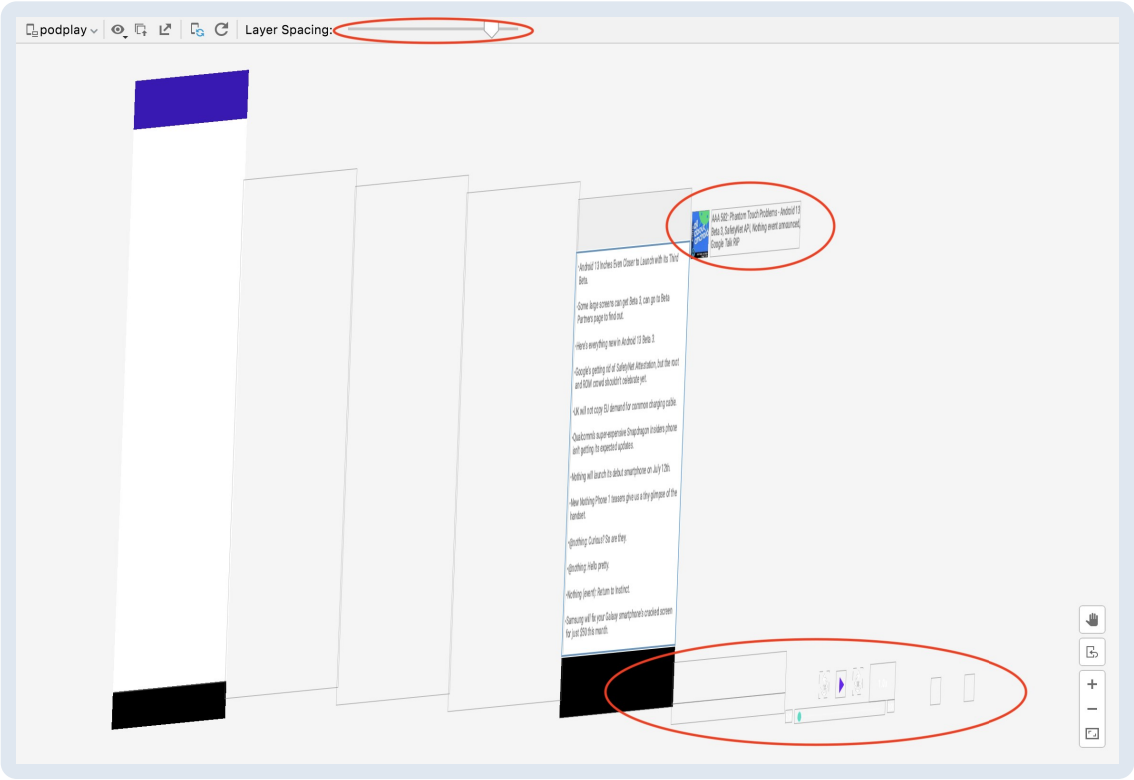
To do this, right-click the **podcastDetailsContainer** and select the **Show Only Subtree** option.

Now, the display section updates to only show the view hierarchy related to the container:



Views in the component tree that aren't a part of the selected subtree will grey out with strikethrough text.

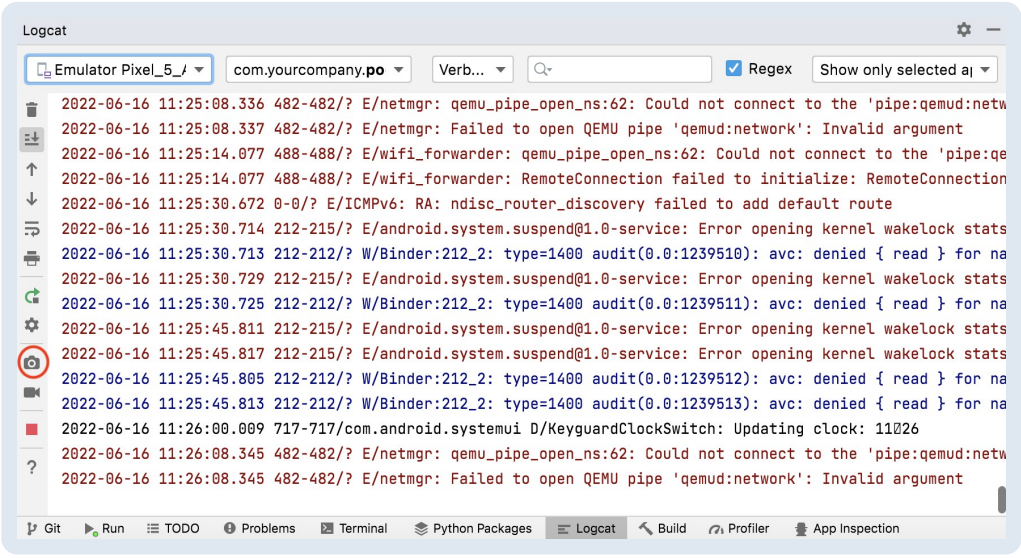
Click and drag the layers in the display section to rotate the 3D model. You can also expand or contract the spacing by adjusting the **Layout spacing** slider. Increase the spacing to provide more clarity in the subtree display:



Take note of the nested layers in the highlighted section in the layer display above. You'll work on condensing this layout to remove some extra layers.

Before you start, take a picture of the app in this state. You'll use this picture later to confirm that your changes align with how the app currently looks.

To take a screen capture, click the **Logcat** tab in Android Studio and click the **Screen capture** icon.



In the dialog window, save the screen capture to your workstation.

Now it's time to improve the layout!

Condensing Layouts With ConstraintLayout

Open **fragment_episode_player.xml** in Android Studio. Look for the `RelativeLayout` with `id` set to `@+id/headerView`:

```
<RelativeLayout
  android:id="@+id/headerView"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:background="#eeeeee"
  app:layout_constraintEnd_toEndOf="parent"
  app:layout_constraintStart_toStartOf="parent"
  app:layout_constraintTop_toTopOf="parent">

  <ImageView
    android:id="@+id/episodeImageView"
    android:layout_width="60dp"
    android:layout_height="60dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:contentDescription="@string/episode_thumbnail"
    android:src="@android:drawable/ic_menu_report_image" />

  <TextView
    android:id="@+id/episodeTitleTextView"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_alignBottom="@+id/episodeImageView"
    android:layout_alignTop="@+id/episodeImageView"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_toEndOf="@+id/episodeImageView"
    android:text="" />
</RelativeLayout>
```

This view is currently a `RelativeLayout` with child views `episodeImageView` and `episodeTitleView`. These child views can move up one level since the root of this layout is already a `ConstraintLayout`. Replace the code above with the following:

```
<ImageView
  android:id="@+id/episodeImageView"
  android:layout_width="68dp"
  android:layout_height="68dp"
  android:paddingStart="8dp"
  android:paddingTop="8dp"
  android:contentDescription="@string/episode_thumbnail"
  android:src="@android:drawable/ic_menu_report_image"
  android:background="#eeeeee"
  app:layout_constraintTop_toTopOf="parent"
  app:layout_constraintStart_toStartOf="parent" />

<TextView
```

```
  android:id="@+id/episodeTitleTextView"
  android:layout_width="0dp"
  android:layout_height="0dp"
  android:paddingEnd="8dp"
  android:paddingStart="8dp"
  android:text=""
  android:gravity="center_vertical"
  android:background="#eeeeee"
  app:layout_constraintTop_toTopOf="@id/episodeImageView"
  app:layout_constraintStart_toEndOf="@id/episodeImageView"
  app:layout_constraintEnd_toEndOf="parent"
  app:layout_constraintBottom_toBottomOf="@id/episodeImageView" />
```

The most notable changes here are:

- Removing of the `RelativeLayout` component.
- Adding background-color `#eeeeee` to each child view.
- Replacing `margin`s with `padding` to properly apply the background color.
- *episodeTitleTextView* is now centered vertically in the parent view using the `android:gravity` property.
- Applying constr aints to each child view.

If you try to compile the app it'll fail because we've removed the **headerView** element which `EpisodePlayerFragment` references. Rather than changing the code to change the visibility of each child view, you can use a `Group`. Add the following below `episodeTitleTextView`:

```
<androidx.constraintlayout.widget.Group
  android:id="@+id/headerView"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"

app:constraint_referenced_ids="episodeImageView,episodeTitleTextVie
w" />
```

This view logically groups `episodeImageView` and `episodeTitleTextView` together under the `headerView` ID. This allows `EpisodePlayerFragment` to reference the same ID without the overhead of an extra view layer in the hierarchy.

The next view group you'll condense is the **playerControls** view. This is a fairly complex `LinearLayout` which contains the controls for the player, such as rewind, play/pause and fast-forward. This is within a nested `RelativeLayout`. There's also another nested `LinearLayout` which holds the time controls, like the start and end times, as well as the seek bar.

Start by replacing the **playerControls** `LinearLayout` with a `ConstraintLayout`:

```
<androidx.constraintlayout.widget.ConstraintLayout
  android:id="@+id/playerControls"
  android:layout_width="0dp"
  android:layout_height="wrap_content"
  android:background="@android:color/background_dark"
  app:layout_constraintBottom_toBottomOf="parent"
  app:layout_constraintEnd_toEndOf="parent"
  app:layout_constraintStart_toStartOf="parent">
```

Remember to change the end tag to match.

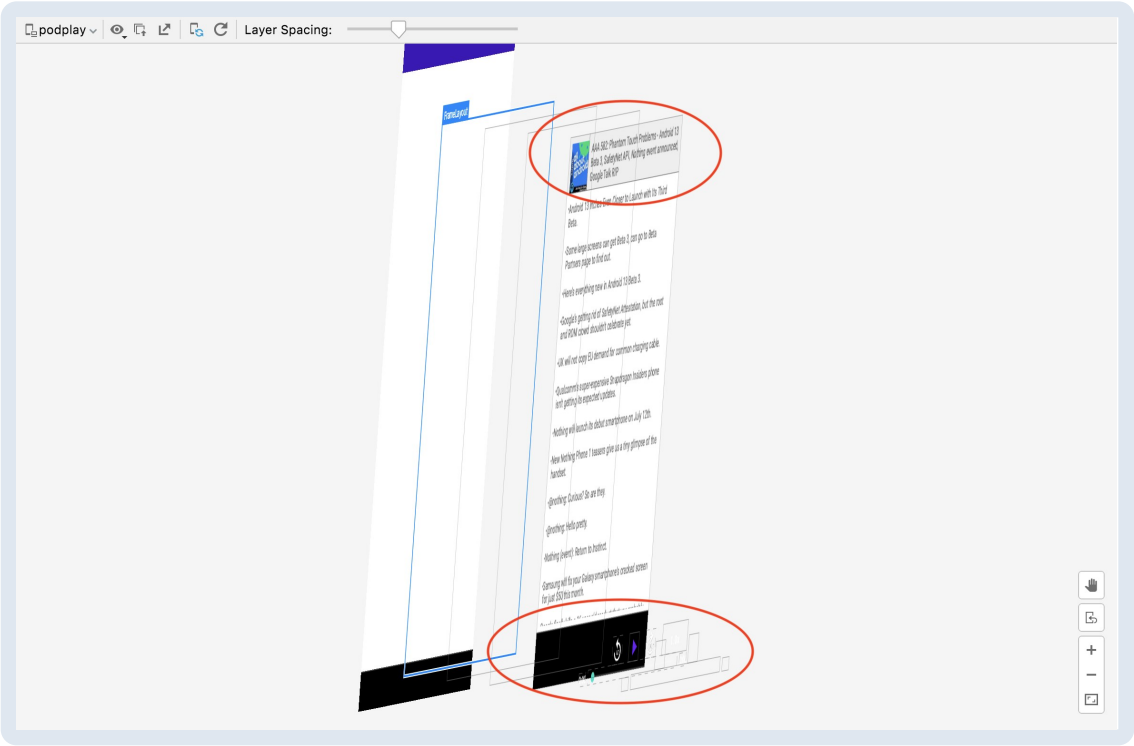
Next, you'll need to add the nested child views as a single, flattened layer. Replace nested `RelativeLayout` and `LinearLayout` views with the following code:

```
<Button
  android:id="@+id/playToggleButton"
  android:layout_width="34dp"
  android:layout_height="34dp"
  android:layout_marginTop="8dp"
  android:background="@drawable/ic_play_pause_toggle"
  android:scaleType="fitCenter"
  app:layout_constraintTop_toTopOf="parent"
  app:layout_constraintStart_toStartOf="parent"
  app:layout_constraintEnd_toEndOf="parent"
  app:layout_constraintBottom_toTopOf="@id/seekBar" />

<ImageButton
  android:id="@+id/forwardButton"
  android:layout_width="34dp"
  android:layout_height="34dp"
  android:layout_marginStart="24dp"
  android:layout_marginTop="8dp"
  android:background="@android:color/transparent"
  android:contentDescription="@string/skip_forward"
  android:scaleType="fitCenter"
  android:src="@drawable/ic_forward_30_white"
  app:layout_constraintTop_toTopOf="parent"
  app:layout_constraintStart_toEndOf="@id/playToggleButton"
  app:layout_constraintBottom_toTopOf="@id/seekBar" />

<ImageButton
  android:id="@+id/replayButton"
  android:layout_width="34dp"
  android:layout_height="34dp"
  android:layout_marginEnd="24dp"
  android:layout_marginTop="8dp"
  android:background="@android:color/transparent"
  android:contentDescription="@string/replay_button"
  android:scaleType="fitCenter"
  android:src="@drawable/ic_replay_10_white"
  app:layout_constraintTop_toTopOf="parent"
  app:layout_constraintEnd_toStartOf="@id/playToggleButton"
  app:layout_constraintBottom_toTopOf="@id/seekBar" />

<Button
```

```
    android:id="@+id/speedButton"
    android:layout_width="wrap_content"
    android:layout_height="0dp"
    android:layout_marginEnd="8dp"
    android:layout_marginTop="5dp"
    android:background="@android:color/transparent"
    android:text="@string/_1x"
    android:textAllCaps="false"
    android:textColor="@android:color/white"
    android:textSize="14sp"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintBottom_toTopOf="@id/seekBar" />

  <TextView
    android:id="@+id/currentTimeTextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="8dp"
    android:layout_marginStart="8dp"
    android:text="@string/_0_00"
    android:textColor="@android:color/white"
    android:textSize="12sp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintBottom_toBottomOf="parent" />

  <SeekBar
    android:id="@+id/seekBar"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginBottom="8dp"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:progressBackgroundTint="@android:color/white"
    app:layout_constraintStart_toEndOf="@id/currentTimeTextView"
    app:layout_constraintEnd_toStartOf="@id/endTimeTextView"
    app:layout_constraintBottom_toBottomOf="parent" />

  <TextView
    android:id="@+id/endTimeTextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="8dp"
    android:layout_marginEnd="8dp"
    android:text="@string/_0_00"
    android:textColor="@android:color/white"
    android:textSize="12sp"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintBottom_toBottomOf="parent" />
```

All the views constrain to either the parent view or each other. No more nested `RelativeLayout` or `LinearLayout` required!

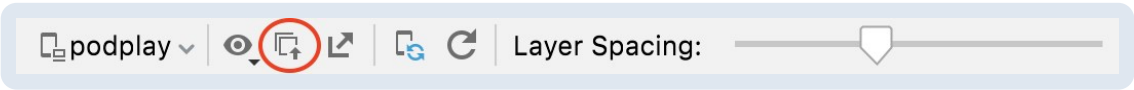
Build and run the app, search “android” again and select the same podcast and episode. You’ll notice that the episode player layout looks the same but with fewer layers. You can also confirm that the layers have been condensed down by using the 3D mode in the layout display:



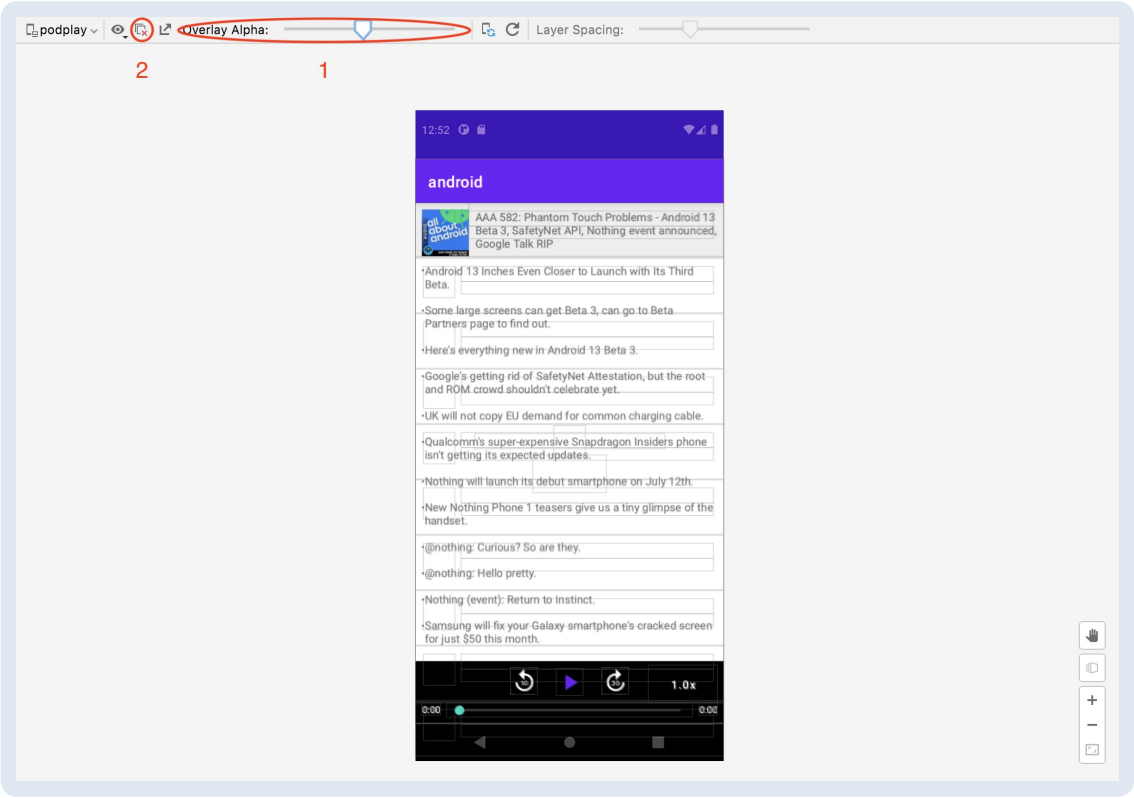
Don't believe your eyes that view layers are the same? Well, next, you'll confirm that they actually do match! :]

Confirming Changes With the Layout Overlay

Start by refreshing the Layout Inspector. Next, click the **Load Overlay** icon:



Select the Logcat screen capture that you saved from earlier. The inspector's display section updates and includes the overlaid image on top of the rendered layout.

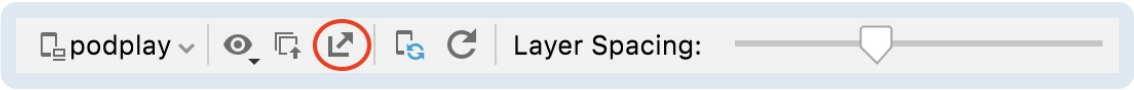


1. Use the **Overlay Alpha** slider to change the overlay's opacity. As you adjust the slider, the image fades in and out. There won't be any changes between the overlay image and the rendering from the Layout Inspector. With this tool, you can officially confirm that the views remained the same while condensing the hierarchy.
2. Finally, click the **Clear Overlay** button in the toolbar.

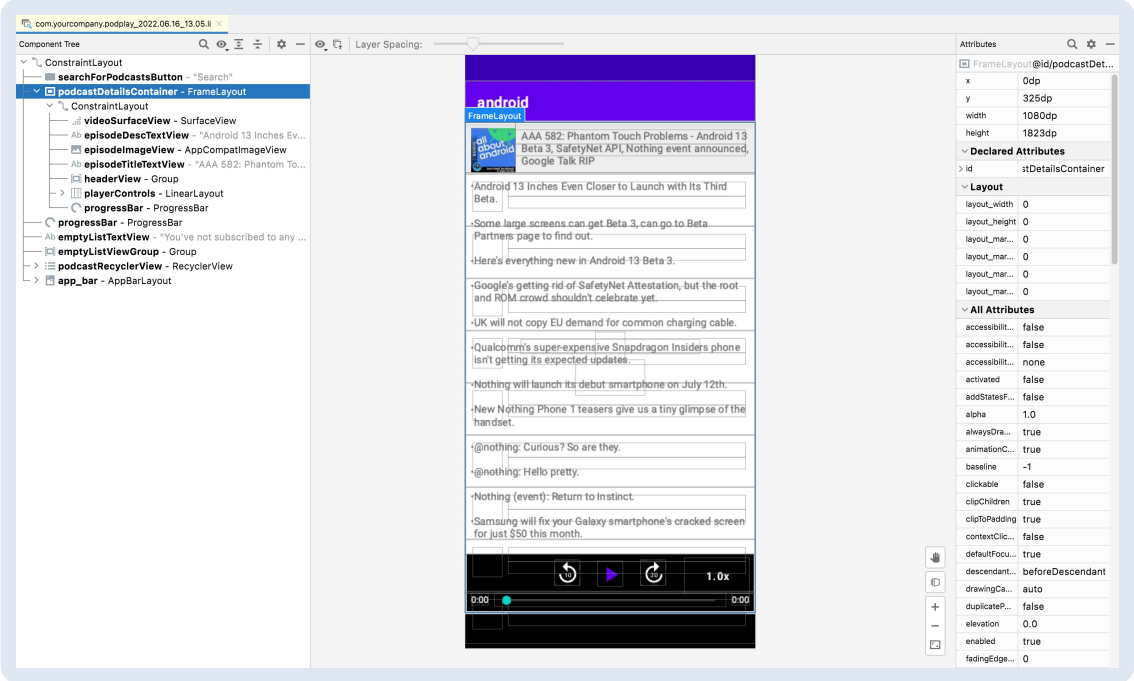
Exporting a Layout Inspector Snapshot

The Layout Inspector provides an export tool where you can save all the inspector details to an **.li** file. This snapshot file is useful for sharing with your team and viewing in Android Studio.

To export an inspector snapshot, click the **Export Snapshot** icon in the toolbar:



When you're ready to view the snapshot, you can open it directly in Android Studio via **File ▸ Open** and select the **.li** file.

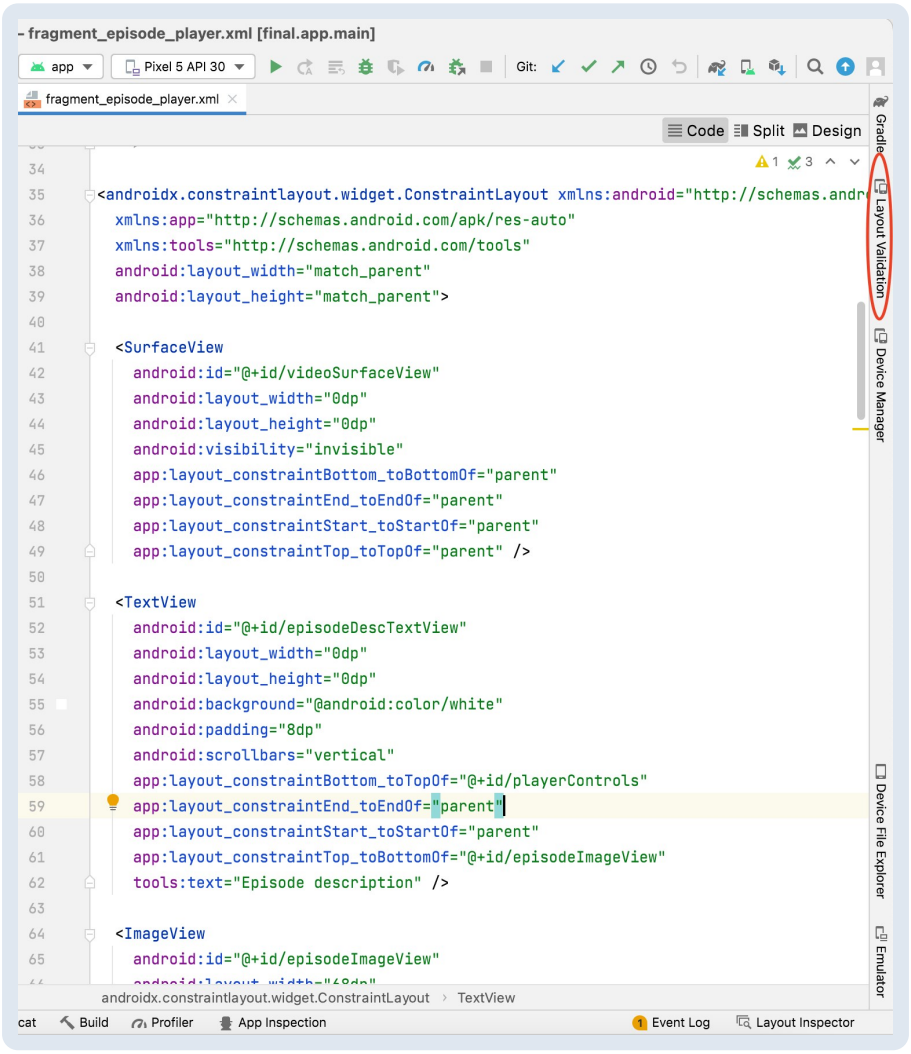


The view for this file is the same as the Layout Inspector window, and you can perform all the same tasks on this screen as you can in the inspector window.

Validating Layouts

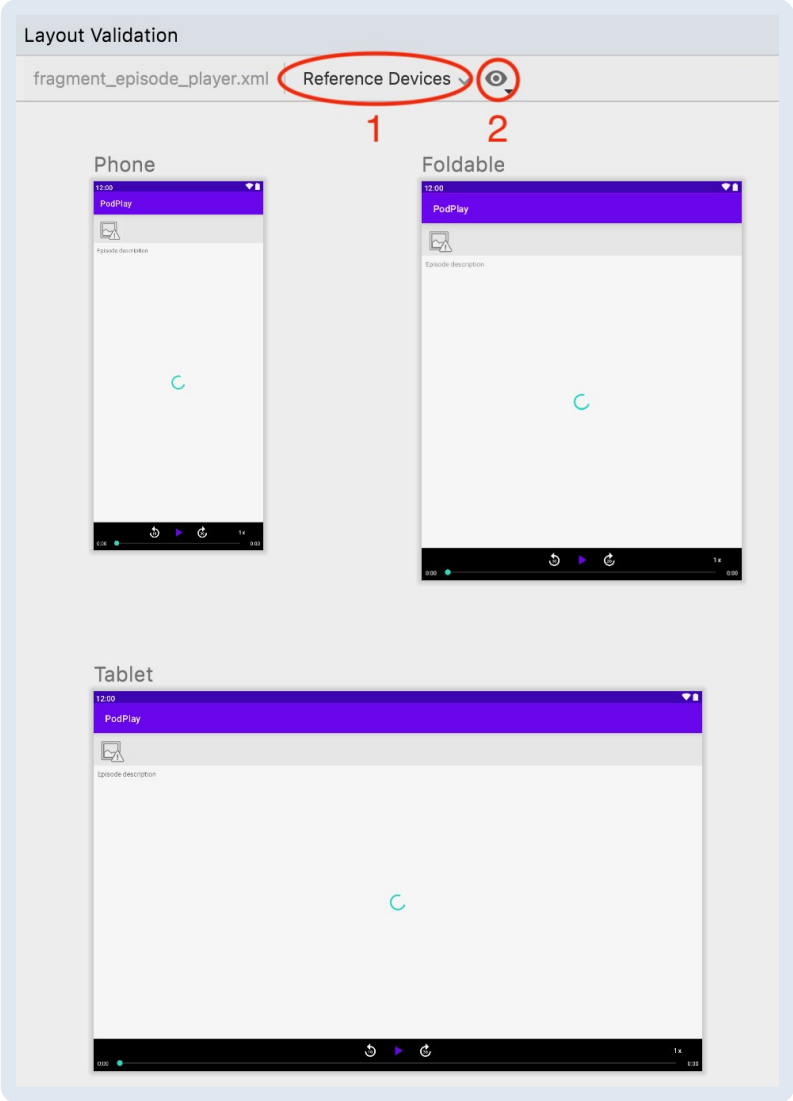
Before running your application, you can visually inspect your changes in the XML layout files with the **Layout Validation** tool. This tool helps you confirm that the layouts you write look correct across a wide range of device references, color settings and font sizes.

Open **fragment_episode_player.xml**, and you'll see that the **Layout Validation** tab on the right vertical toolbar appears.




```
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.andr
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <SurfaceView
    android:id="@+id/videoSurfaceView"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:visibility="invisible"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

  <TextView
    android:id="@+id/episodeDescTextView"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:background="@android:color/white"
    android:padding="8dp"
    android:scrollbars="vertical"
    app:layout_constraintBottom_toTopOf="@+id/playerControls"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/episodeImageView"
    tools:text="Episode description" />

  <ImageView
    android:id="@+id/episodeImageView"
```

Select this tab to open the validation window:



This window shows you what the layout will look like under different device configurations.

Two notable options in the window toolbar are:

1. **Configuration Set**: Changes the device configuration itself. This is where you can choose different device types and do colorblind tests or font size validation.
2. **View Options**: Provides options to change the layout. The main option is **Show System UI**, which you can enable now. This particular option will display the status and action bars at the top and the hardware buttons at the bottom.

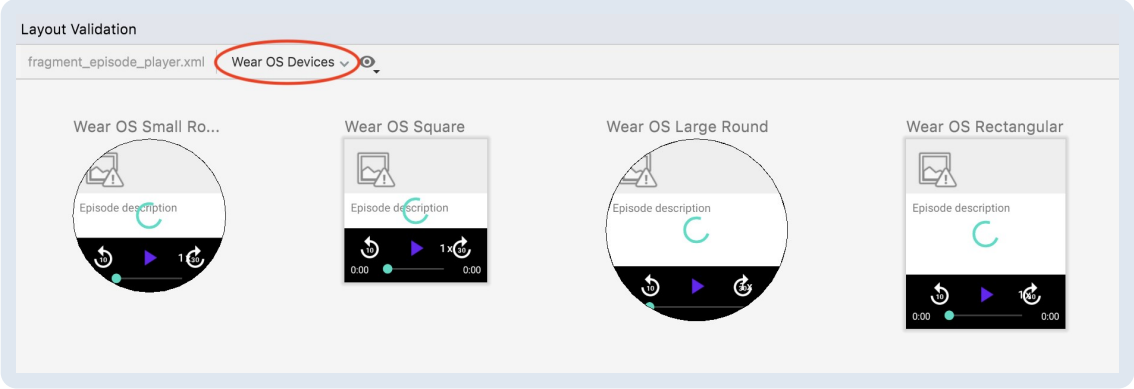
Devices Types

As shown above, the default configuration setting is the **Reference Devices**. Using this configuration, the Layout Validation shows the layout on a range of device types, including phones, foldables, tablets and even desktop

configurations. This saves you time from having to load multiple emulators for each device type or gather actual devices and run the app on each one.

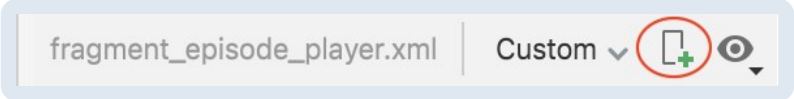
You can also show Wear OS configurations if you're working on a wearable app. While the episode player view isn't intended for this use, you can still check out how the layout will appear on the different Wear OS configurations.

Change the configuration set to **Wear OS Devices**:

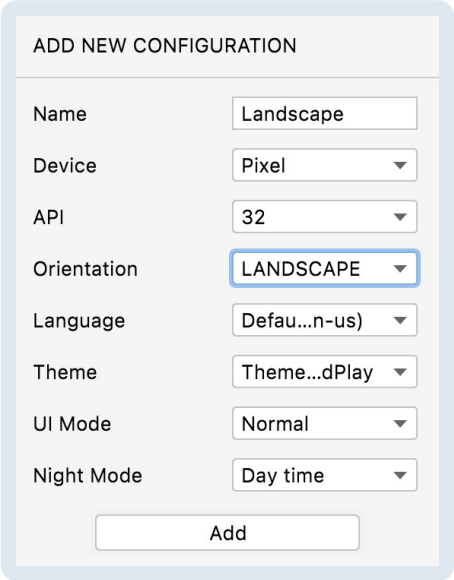


Custom Configurations

You can create a custom configuration based on the criteria that you want to validate. To do this, change the configuration set to **Custom**. Notice that the validation toolbar updates to include the **Add configuration** icon.

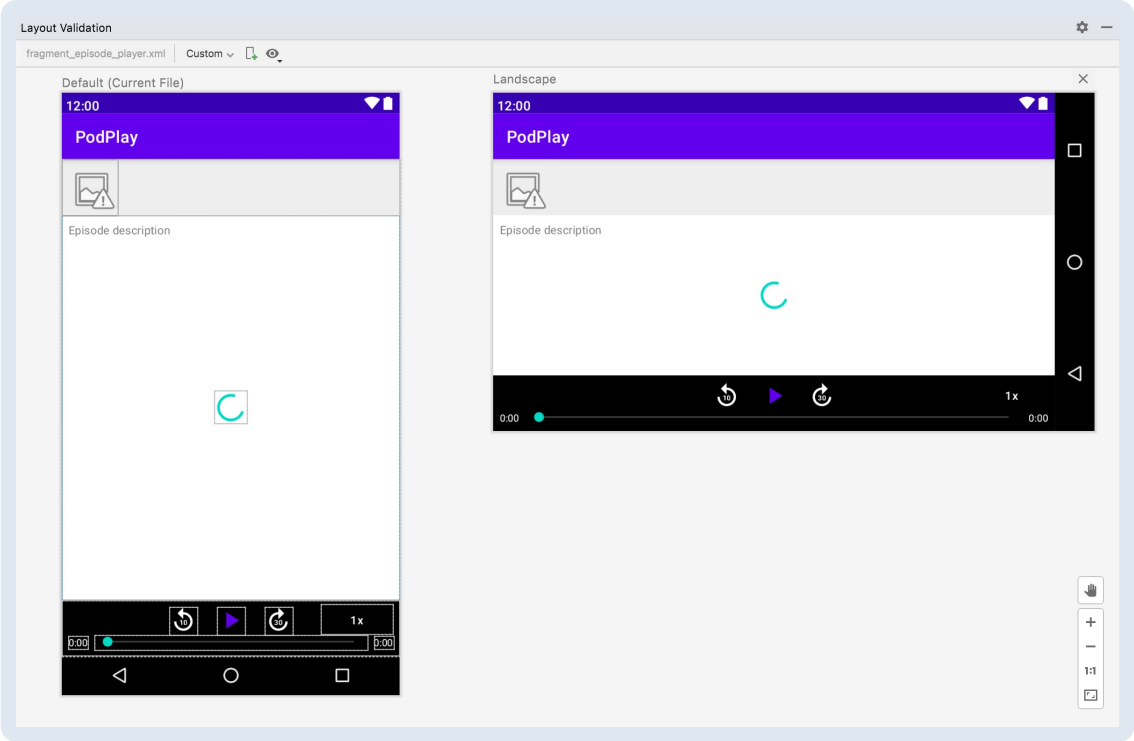


Click the icon and a dialog appears:



Here, you can set your configuration settings. For this example, give it the name "Landscape" and choose **LANDSCAPE** for the orientation.

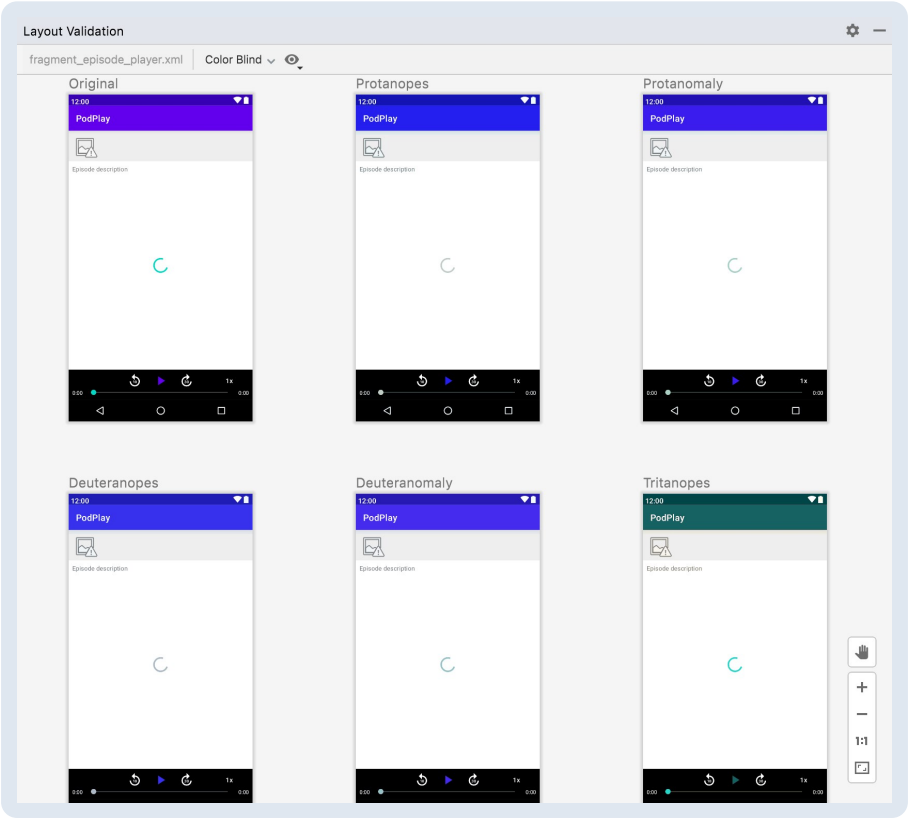
Click **Add**, and the validation window will update to include the layout using your custom configuration:



Colorblind Tests

It's important to ensure your layouts are accessible to as many people as possible. In the Layout Validation tab, you can perform a layout test to confirm that your app is still accessible to users who are color blind.

Change the configuration set to **Color Blind**:



Multiple layouts appear showing the different types of common color blindness:

- **Protanopes** and **Protanomaly** represent how people with severe and mild red-green color blindness will see the layouts. Defects in the red cone eye pigments of the eye cause this type of color blindness.
- **Deuteranopes** and **Deuteranomaly** show how the layouts appear to those with severe and mild red-green color blindness caused by defects in the green cone pigments.
- **Tritanopes** represents the layouts in the way users with blue-yellow color blindness see them.

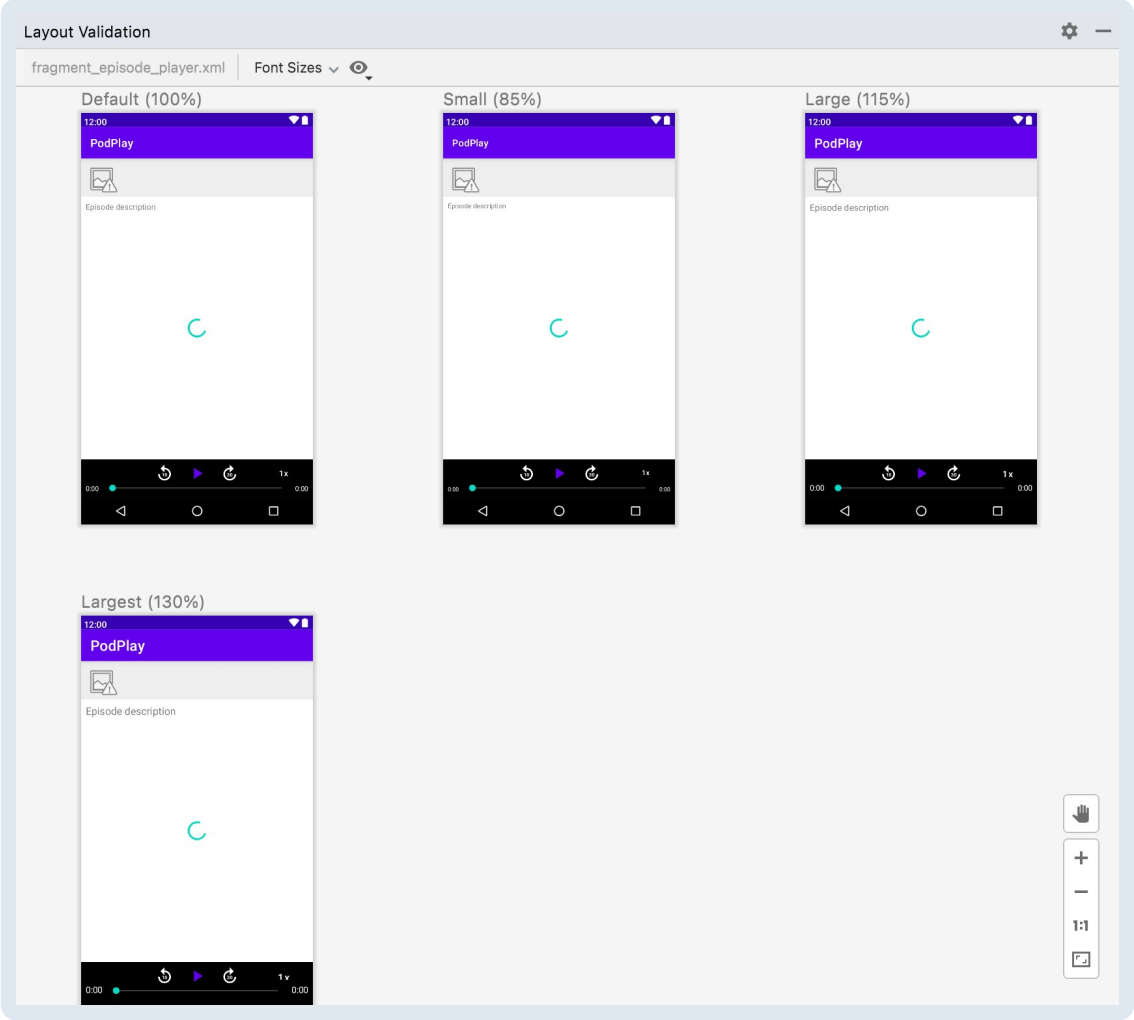
These different views help you determine if there are color clashes that you'll need to address before releasing your app to the public.

Font Sizes

Along with the topic of accessibility, font sizing is another essential aspect to test. A common way of testing your app is by adjusting the font size via the Settings app and then opening your app and verifying that the changed font size looks good.

With the **Font Sizes** configuration, you can easily view at once how your layout will appear with different font sizes.

Change the configuration set to **Font Sizes**:



The default font size at 100% is displayed, along with small — 85%, large — 115% and largest — 130%. This allows you to see font size differences side-by-side. You can also determine if there'll be any issues with your layout if a user needs to use an increased font on their device.

Challenge

Congratulations on condensing down those views in the episode player! But, there's still more you can do. You converted the **playersControl** view to a `ConstraintLayout` in **fragment_episode_player.xml**. Improve that layout again and remove the **playerControls** layer altogether. After removing the layer, move its children into the root `ConstraintLayout`. Afterward, verify that the layer is gone by using the Layout Inspector and its 3D rendering mode.

Key Points

- The Layout Inspector displays the views as layers in the UI hierarchy.
- The inspector's 3D render tool allows you to view the layers.
- Add overlays in the inspector to confirm changes are accurate.
- Export snapshots of the inspector for later use.
- Validate layouts against device types, color blindness tests, and font sizes.

Where to Go From Here?

To learn even more about the Layout Inspector, read the associated Android developer user guide.

If you're interested in learning more about accessibility, check out Android Accessibility by Tutorials.

The journey isn't over yet! In the next chapter, you'll learn how to live query databases with the **Database Inspector**.

7 Debugging Databases

Written by Zahidur Rahman Faisal

Data is a collection of facts, such as numbers, words, measurements, observations or just descriptions of things. - Math is Fun

The word *Data* originates from the word *datum* which means a single piece of information. Data is the plural for datum.

Basically, **data** is a distinct, small unit of information that can be used in various forms like text, numbers, media or bytes etc. It can be stored in electronic memory or pieces of paper just like in a book!

In the previous chapter, you learned about debugging views and optimizing layouts to display details of the podcast channel that you've subscribed. At this point, you must be wondering if you could debug the way podcasts are stored in PodPlay.

In this chapter, you'll check how your subscribed podcasts are stored and organized in your database, and you'll resolve the episode ordering issue.

In this process, you'll learn how to:

- Explore and update your existing data.
- Modify your queries and watch live updates.
- Export your Database.

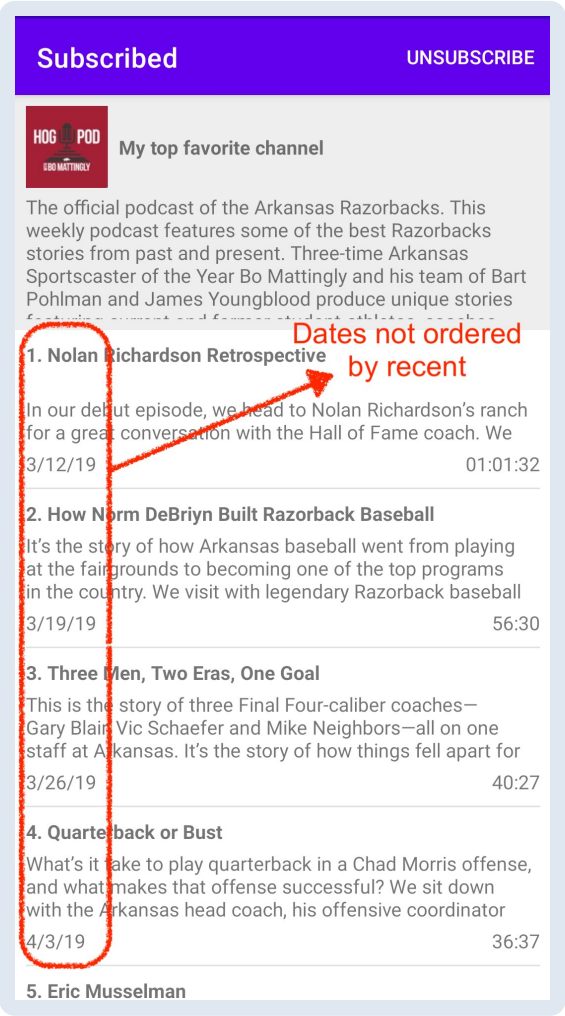
Searching for The Data

Imagine a library that consists of many books where you can go and easily find a book about a topic you're interested in or store a new book to read later. That's your Database.

It wouldn't be easy for you to find a book in a library if the books weren't organized. You might get lost searching for a book that's recently been added if someone just put the book randomly on any shelf. You might have faced the same situation in the **PodPlay** app! When using the app, you access a vast library of podcasts and can subscribe to your favorite ones to come back to later. The details of your subscribed podcast channels are stored locally in the app, just like a library. PodPlay uses a **Room Database** to store your subscriptions.

To start, open the **Podplay** starter project and run the app. Tap the search icon in PodPlay, type "RW" in the search bar and press **Return**. Choose any podcast and tap it to open its details. Finally, tap **SUBSCRIBE** to subscribe to the podcast.

You'll now see the channel name in the subscribed channel list. Tap it to open a detailed screen, and you'll see a list of episodes underneath the title and description of the channel. If you look closely, you may notice the episodes aren't shown in the correct date order; you'd expect to see the recent episode on top of the list and then the older ones in chronological order.



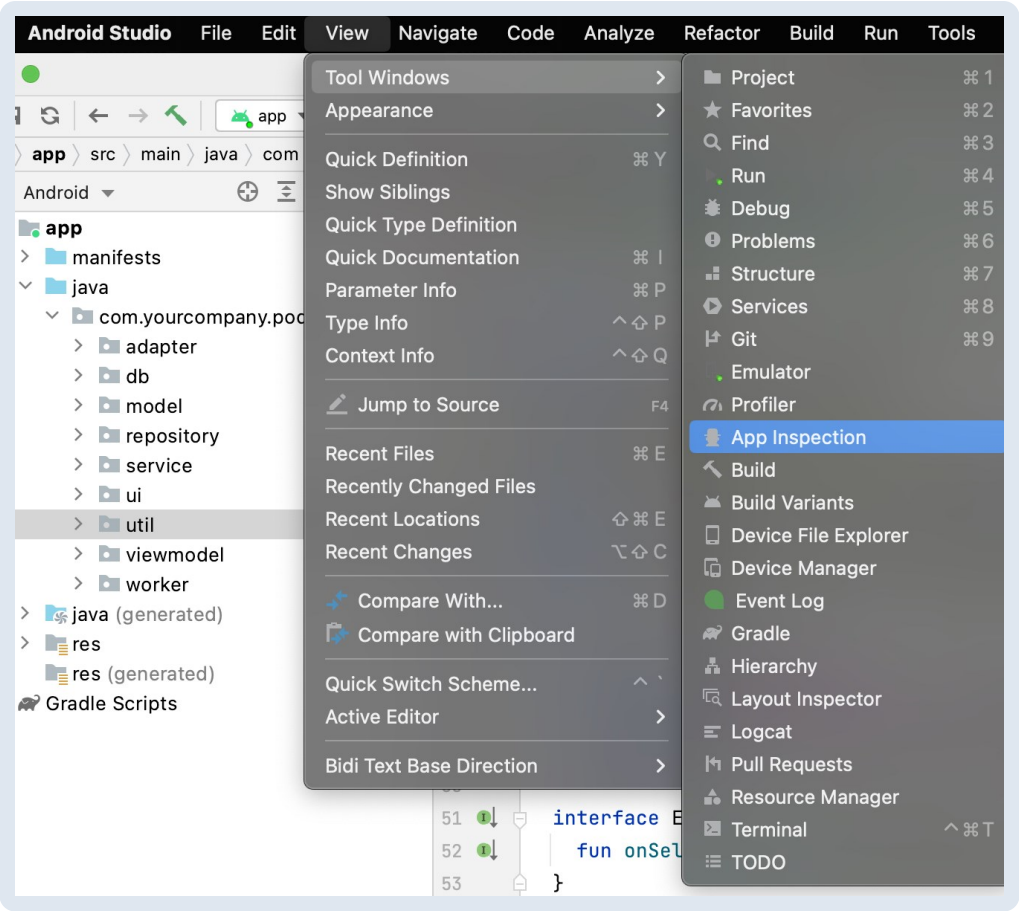
There's an integrated **Database Inspector** in Android Studio 4.1 and higher to assist your inspection of what's gone wrong. The Database Inspector allows you to easily inspect, query, and modify the database in PodPlay, just like editing a spreadsheet; who wouldn't want to take advantage of that!

The Database Inspector

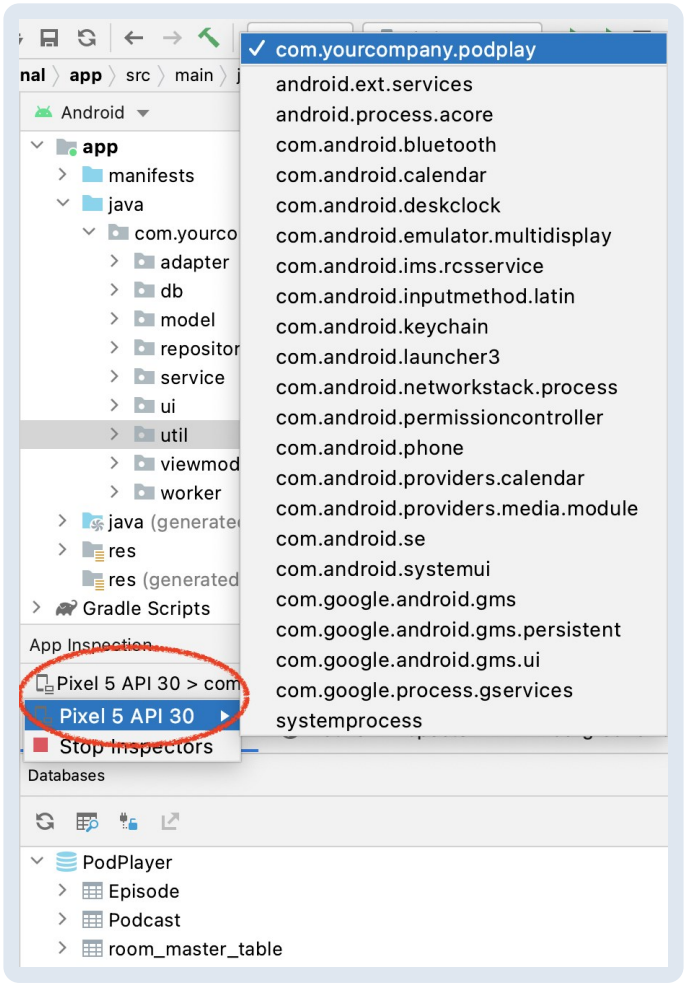
You might be questioning how PodPlay's database arranges your subscribed podcasts. Soon you'll be able to see your table structure and schema using the

Database Inspector. To open the Database Inspector in Android Studio, you need to:

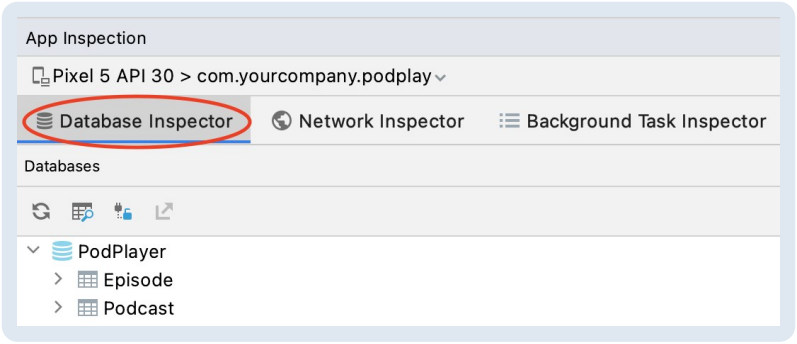
1. Run your app on an emulator or connected device running API level 26 or higher.
2. Select **View ▸ Tool Windows ▸ App Inspection** from the menu bar.



3. Choose **com.yourcompany.podplay** from the running app process in the drop-down menu.



4. Then Select the **Database Inspector** tab.



Previewing Database

You'll see a list of databases in your app and the tables each database contains within the Databases pane. The name of the database in PodPlay is **PodPlayer**.

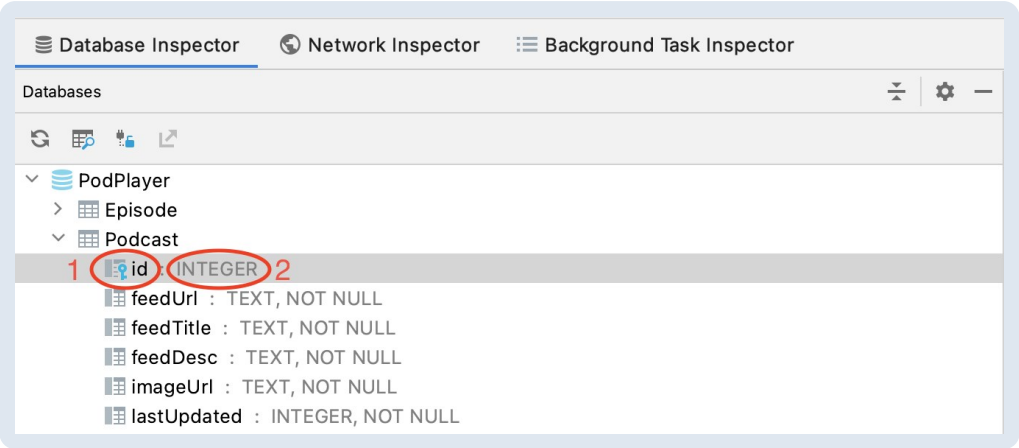
The PodPlayer database consists of two tables as follows:

1. **Episode**: A list of episodes in subscribed channels including additional fields, such as `mediaUrl`, `type`, `duration`.
2. **Podcast**: Mainly contains the podcast URL, name and description of a subscribed podcast channel.

Simply double-click on the **Podcast** table to display its data in the inspector window.

Find the Schema

Here each column represents a field in your table. Expand the Podcast table to see how the schema was defined:



The marked area in red provides two important pieces of information:

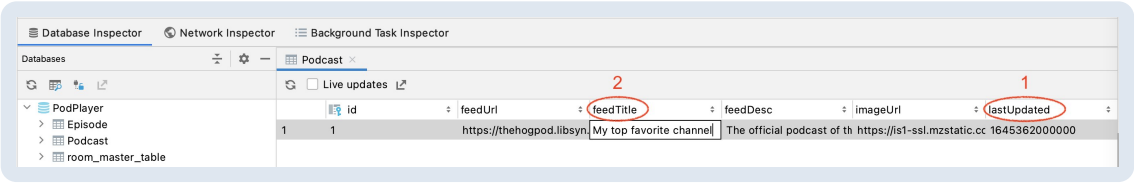
1. `id` is the **primary key** for this table.
2. The data type for the primary key is **Integer**.

Looking at this, you can verify if you've defined the proper data types for your fields. This is useful to tackle issues where any unsupported data might have accidentally been inserted into a specific field in your table.

Updating Existing Data

Start by verifying if the data is in the correct order and getting stored in your table properly by following two simple steps:

1. To find your latest added podcast, click the `lastUpdated` column. It's the timestamp of when you subscribed to the channel. Clicking on a column header sorts the data in the inspector window by that column, so it'll display the rows in a chronological order based on your timestamp.
2. Double-click on the first item under the `feedTitle` column, then change the title to "My top favorite channel" and press **Return**.

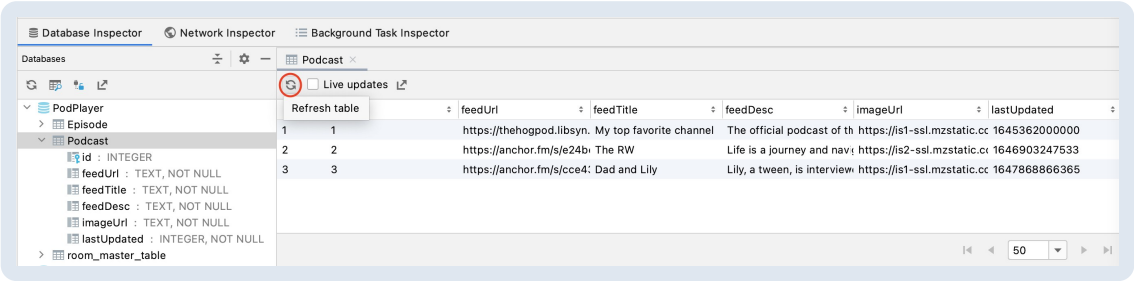


PodPlay uses Room Database, and the UI observes the database with **LiveData**, so your update will be instantly visible in your running app. Otherwise, the changes would only be visible after you launch the app next time.

Observing Live Updates

So far, it seems the data in the Podcast table has been stored and retrieved as expected. Now take it one step further, type something in the search bar, select a channel and tap **SUBSCRIBE**. It's supposed to be on your Podcast table, but you can't see any change!

This is because you need to reload the contents of the current table that's visible to you in the Database Inspector.



Now, click the highlighted icon in the above image. This will refresh the contents, and the channel you just subscribed to will show up in the Podcast table!

It's convenient to see live updates while you interact with a running app; to enable that feature you need to click the **Live updates** checkbox beside the Refresh icon.

Enable the Live Updates feature and subscribe to something. The subscribed channel is immediately shown in your Podcast table, as the name suggests.

Note that the table in the inspector window becomes *read-only*, and you can't modify its values with **Live updates** enabled. So, trying to double-click and change a value will have no effect while in **Live updates** mode.

Querying Your Databases

Since you verified that the subscribed podcasts are getting stored and retrieved

correctly, the next step is to check if your database queries are correct.

The Database Inspector can run queries on your app's database while you're running the app. The tool can easily execute DAO queries that you've written for your Room database, but it also allows you to write custom SQL queries to validate your ideas!

Running DAO Queries

Data Access Object (DAO) is an interface that executes CRUD operations in your database, hiding the complexity behind it.

Open **PodcastDao.kt** within the **db** package in your project folder; `PodcastDao` is a simple DAO used to interact with the database in PodPlay.

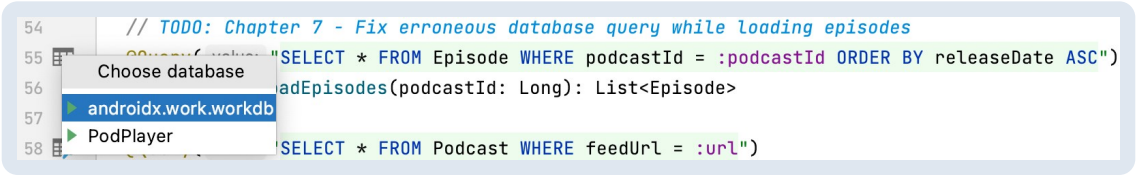
Looking at the queries here, it's obvious that `loadEpisodes()` is responsible for providing a list of episodes to your UI layer.

The `podcastId` parameter is the `id` of your podcast channel from the **Podcast** table. The query filters out episodes from the entries in the **Episode** table based on the provided `podcastId` here.

You can easily find out what this query returns by clicking on the button next to `@Query` like below:

```
46  @Dao
47  interface PodcastDao {
48    @Query( value: "SELECT * FROM Podcast ORDER BY FeedTitle")
49    fun loadPodcasts(): LiveData<List<Podcast>>
50
51    @Query( value: "SELECT * FROM Podcast ORDER BY FeedTitle")
52    suspend fun loadPodcastsStatic(): List<Podcast>
53
54    // TODO: Chapter 7 - Fix erroneous database query while loading episodes
55    @Query( value: "SELECT * FROM Episode WHERE podcastId = :podcastId ORDER BY releaseDate ASC")
56    suspend fun loadEpisodes(podcastId: Long): List<Episode>
57
58    @Query( value: "SELECT * FROM Podcast WHERE feedUrl = :url")
59    suspend fun loadPodcast(url: String): Podcast?
60
61    @Insert(onConflict = REPLACE)
62    suspend fun insertPodcast(podcast: Podcast): Long
63
64    @Insert(onConflict = REPLACE)
65    suspend fun insertEpisode(episode: Episode): Long
66
67    @Delete
68    suspend fun deletePodcast(podcast: Podcast)
69  }
```

Upon clicking, a prompt appears for you to select the database to execute the query on. Select **PodPlayer** from the drop-down list.

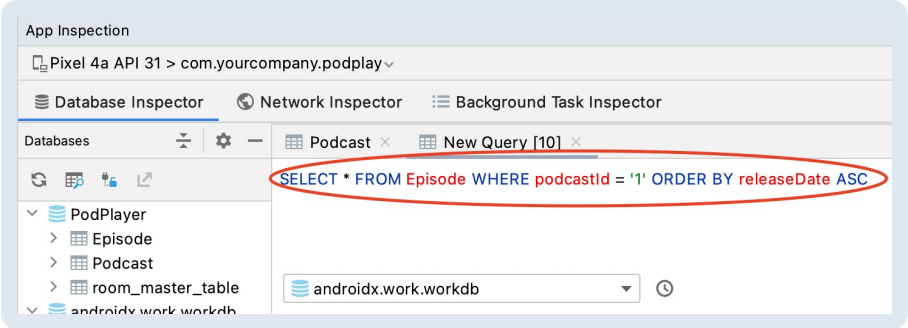


Next, there will be another popup showing the query you choose to execute and asking for a value for your query parameter, if any, which is `podcastId`.

Input “1” to load episodes from your first subscribed podcast channel.

If your query method includes more than one parameter, Android Studio requests values for each parameter before running the query.

You’ll see the results of the query in a new tab within the Database Inspector like below:

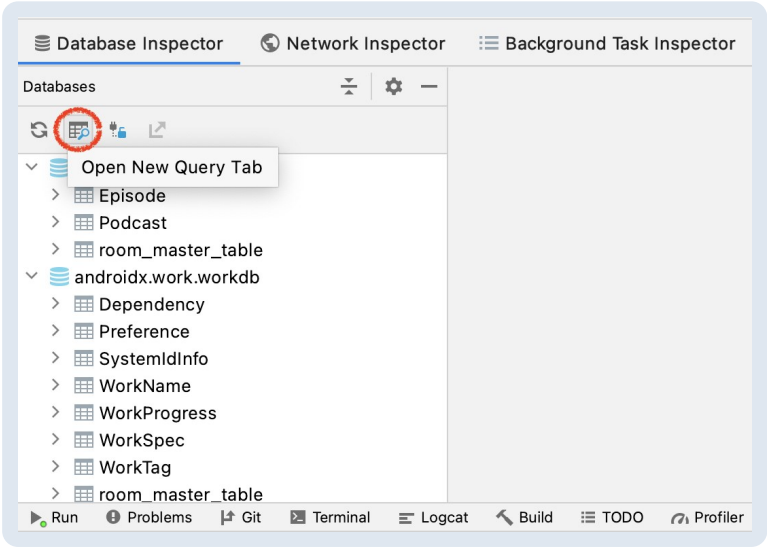


Running Custom Queries

Focus on the highlighted area in your query above; you’ll see your results in *ascending* order of each episode’s `releaseDate`. This must be why you don’t see the latest episode at the top of your list!

To fix this, you need to have the episode listed in *descending* order by `releaseDate` when you execute this query. You can quickly verify this by running a custom query without even modifying your DAO file by following the below steps:

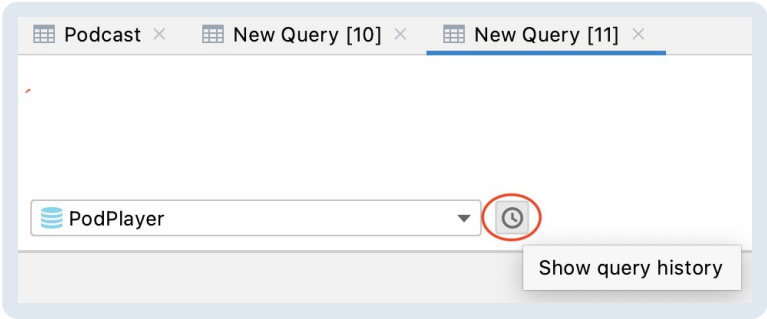
1. Open the **Open New Query** tab at the top of the Databases pane.



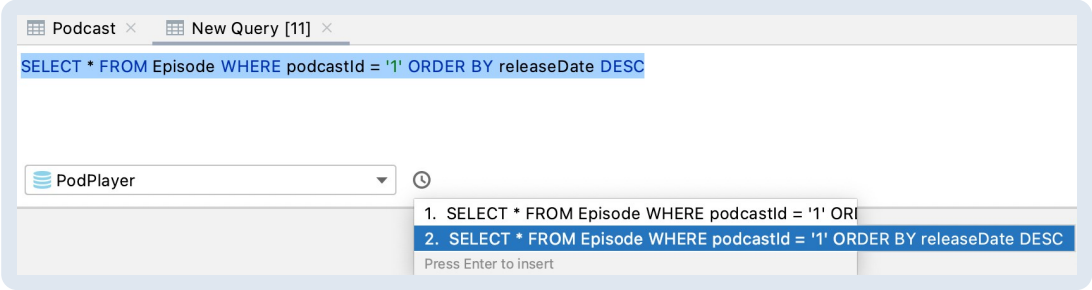
2. Select the database **PodPlayer** to query from the drop-down list on the New Query tab.
3. Type the following SQL query into the text field at the top of the New Query tab: "SELECT * FROM Episode WHERE podcastId = '1' ORDER BY releaseDate DESC"
4. Click **Run** to execute the query.

Here's another simple approach to doing this since your query hasn't changed much apart from a keyword:

1. Click the **Show query history** icon beside the database name in your custom query pane to see a list of queries that you previously ran.



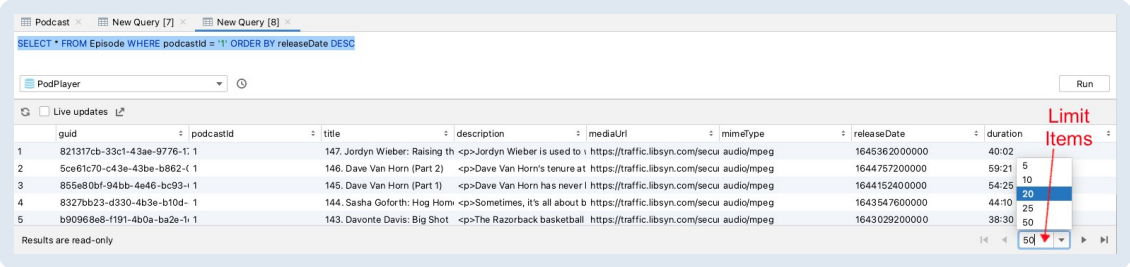
2. Click a query in the list to see a preview of the full query in the editor, and press **Return** to copy it to the editor.
3. Modify the query part you want. In this case, change the order from **ASC** to **DESC** in your query.



4. Click **Run** to execute the statement.

Look at the `releaseDate` column now; the items are in descending order. That means the custom query will return the latest episode first and the others subsequently, which solves your issue!

You can limit the number of items on a page for simplicity by clicking on the highlighted area.



Now you're certain the new query works and it will deliver your expected results, open **PodcastDao.kt** and update the `@Query` above `loadEpisodes()` like below:

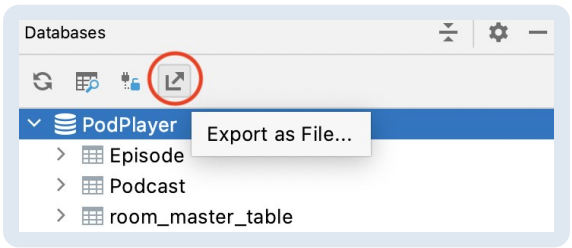
```
SELECT * FROM Episode WHERE podcastId = '1' ORDER BY releaseDate
DESC
```

With this code, you change the order of podcasts.

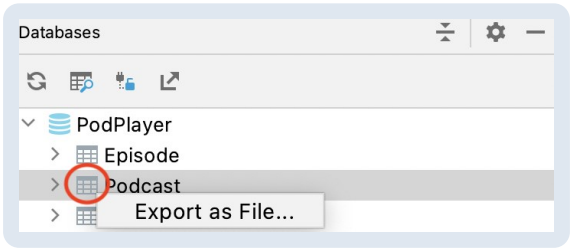
Exporting Data from the Database Inspector

The Database Inspector allows you to export databases, tables or query results. It's a powerful feature to save, share or replicate your data. You can export using any of the following ways:

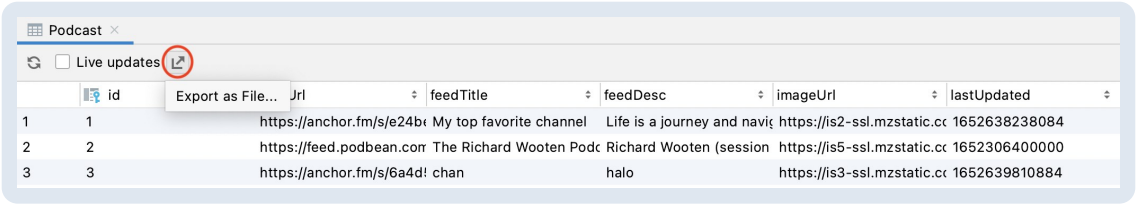
- Select a database in the Database pane, then click **'Export as File...'** near the top-left corner to export the whole database.



- Right-click on a table in the Databases panel and select **'Export as File...'** from the context menu to export a specific table in the database.



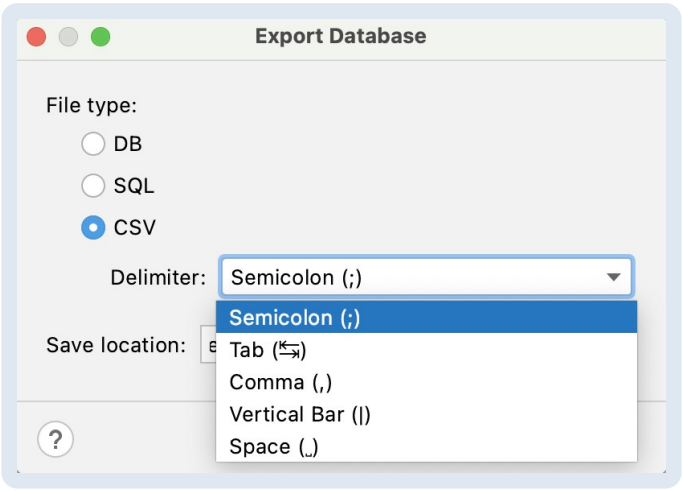
- Click **'Export as File...'** query results while viewing a table or query results in a tab.



To complete the export action, you need to choose a format for the exported file. Depending on whether you're trying to export a database, table, or query results, the Database Inspector will popup three options for exporting the data in the following formats:

1. **DB:** Preferable format if you're going to import or use the database in mobile applications.
2. **SQL:** Easy to open or modify the exported file on any desktop or web-based database system.
3. **CSV:** Simple, lightweight and widely accepted format, good for batch reading/data writing.

If you select CSV format, you can also choose how the values are separated in a row from the drop-down below:



Then the Database Inspector will accumulate the tables from the PodPlayer database in different **.csv** files and create a single **.zip** on your selected path to export.

This feature can save your day in an emergency by simplifying the backup and sharing process. You can quickly restore your entire database with a few clicks!

Key Points

- You can look into your existing database easily with Database Inspector.
- Check your database schema carefully and validate your assumptions about updating fields.
- Using the Live updates feature will save you time to observe changes, but you can't modify data in this mode.
- Save time running your queries directly from the Android Studio DAO.
- Copy and modify a query or write a custom one to see quick results.
- Export your database for easy backup or sharing your data.

Where to Go From Here?

You're now an expert on Database Debugging! The Database Inspector offers you other powerful features such as:

- Offline mode
- Keeping database connections open

If you're interested in mastering the ways of storing data on Android, these are the best starting point for you:

- Saving Data on Android
- Room Database Tutorial

At this point, you're probably curious to know how fetching and saving data works in the background! In the next chapter, you'll learn to inspect on background tasks using **Background Task Inspector**, so stay tuned!

8 Debugging WorkManager Jobs With Background Task Inspector

Written by Zahidur Rahman Faisal

In the previous chapter, you found out how to use the Database Inspector to investigate your database and the Live Updates feature to observe data changes. You learned how to use a query for previewing results while debugging the app and you got familiar with exporting your database to make sharing easier.

All this sounds really great, but you're probably wondering where to find the data for populating your local database and how to download it. That could become a demanding task if the data is big and you need to sync it often. You need to think about user experience as well. Therefore, the solution is to run complex tasks in the background.

Performing long-running operations and processing data in the background is very common for mobile applications. Whether syncing your everyday tasks in the calendar or backing up your contacts and images to the cloud, they're all **background tasks**, a basic and necessary part of your smartphone usage habit.

Here are the conditions one task needs to meet to become a background task:

- None of the ongoing activities are currently *visible* to the user.
- The task isn't running any *foreground services* that the user explicitly started.

Common Types of Background Tasks

A background task includes additional processes like scheduling, monitoring, logging and user notification separately from the actual work it's supposed to do. Considering the type of work it does, a background task can be any of the three types below:

1. **Immediate:** Expedited tasks that must begin immediately and complete quickly, e.g. fetching data from the server.
2. **Long-Running:** Tasks that run for a more extended period, usually more than 10 minutes, e.g. downloading big files.
3. **Deferrable:** Scheduled tasks that start at a later point in time or run periodically, e.g. periodic sync with the server.

PodPlay relies on Android WorkManager to handle all types of background tasks. The app is smart enough to regularly check your subscribed channels and update the episode list if new episodes are published. This job runs in the background without you even realizing it's happening!

This is great, but it can take a toll on your mobile data and CPU usage if the background operation is frequently running. In this chapter, you'll inspect background tasks and schedule them to use optimal resources. You'll learn how to:

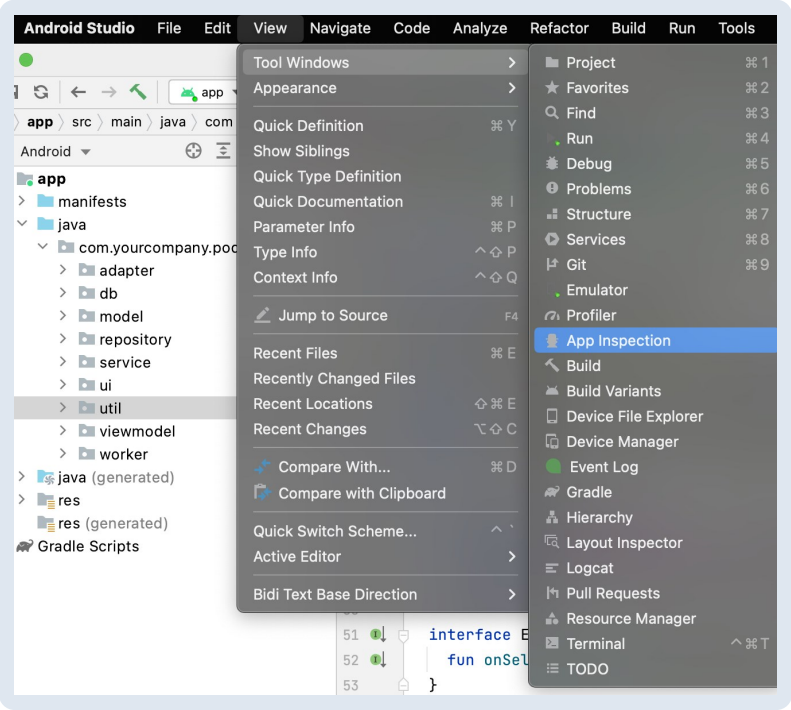
- List currently running or scheduled background tasks.
- Find details of a background task.
- Check the execution flow of background tasks.

Starting the Background Task Inspector

The scheduling of background tasks happens when you launch PodPlay. You can then see what's happening by starting the **Background Task Inspector**.

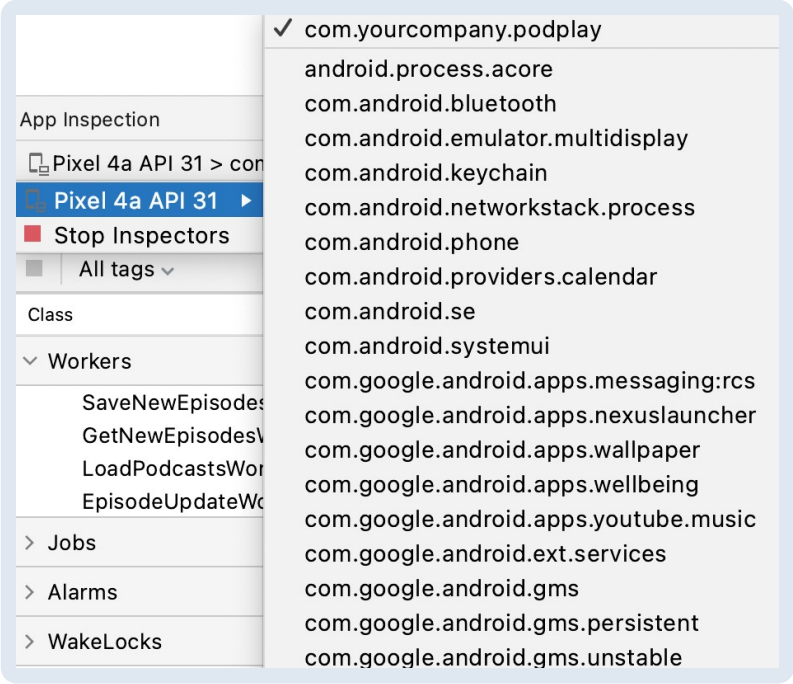
Here are the steps:

1. Open the **Podplay** starter project and run the app on an emulator or connected device using API level 26 or higher.
2. Select **View ▸ Tool Windows ▸ App Inspection** from the menu bar.

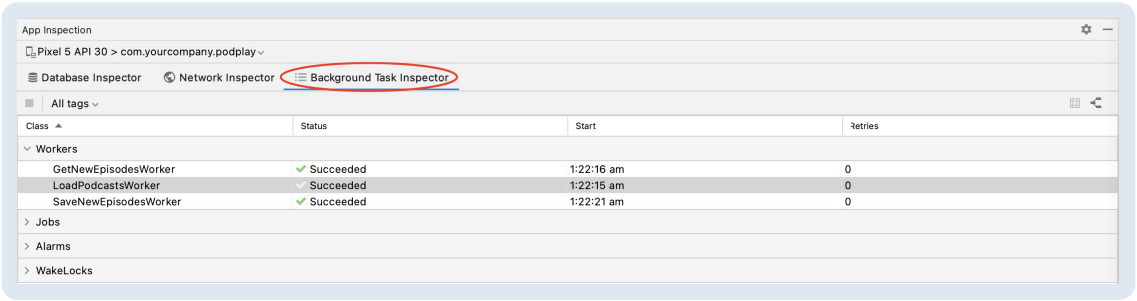


3. In the App Inspection toolbar, choose your device. Then, select **com.yourcompany.podplay** from the running app process in the dropdown

menu.



4. Then select the **Background Task Inspector** tab.



Now, you prepared the environment for inspecting PodPlay.

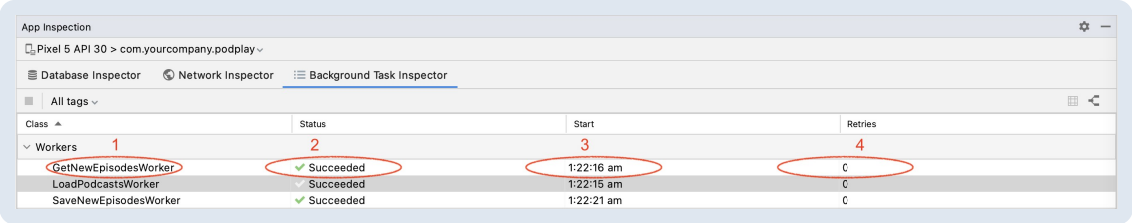
Viewing and Inspecting Workers

PodPlay contains three different workers that check for new episodes in your subscribed channels:

1. **LoadPodcastsWorker** loads all podcast channels saved in `PodcastRepo`.
2. **GetNewEpisodesWorker** fetches the list of episodes for each individual channel from the server, then compares it with existing data in `PodcastRepo` to find if there are new episodes.
3. **SaveNewEpisodesWorker** appends in `PodcastRepo` if new episodes are found and notifies the user.

The Work Table

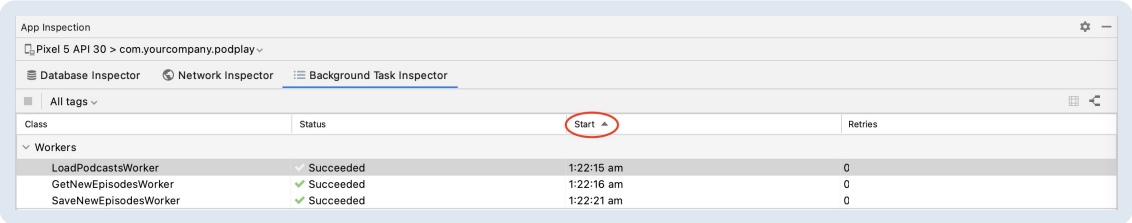
Background Task Inspector displays a table listing all the tasks which `WorkManager` assigns.



At first glance, it reveals four important pieces of information about a worker:

1. The class name that implements the `Worker` interface handling the work.
2. What state that work is in right now (e.g., Enqueued, Running, Blocked, Succeeded or Failed).
3. The exact time that the work started execution.
4. How many times the work has been retried if there was any problem during execution.

The table displays the list of workers in alphabetical order based on their class name by default, but you can sort them based on each field above. For example, click the **Start** column, and that task will be sorted by its start time, as you can see below:



Extracting Work Details

The table does a lot more than just show you a list of works; it allows you to filter each individual work by its tag and extract task details.

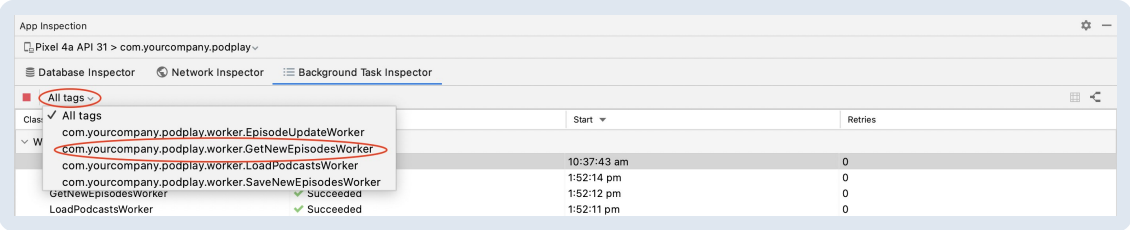
Your next objective is to leverage them and find details about a specific worker from the `WorkManager`'s queue.

Finding a Work by Tag

When your `WorkManager` is executing a whole lot of work, the easiest way to find one of them from the table is to use the **Tag Filter**.

Click **All Tags**, then select **com.yourcompany.podplay.worker.GetNewEpisodesWorker** from the

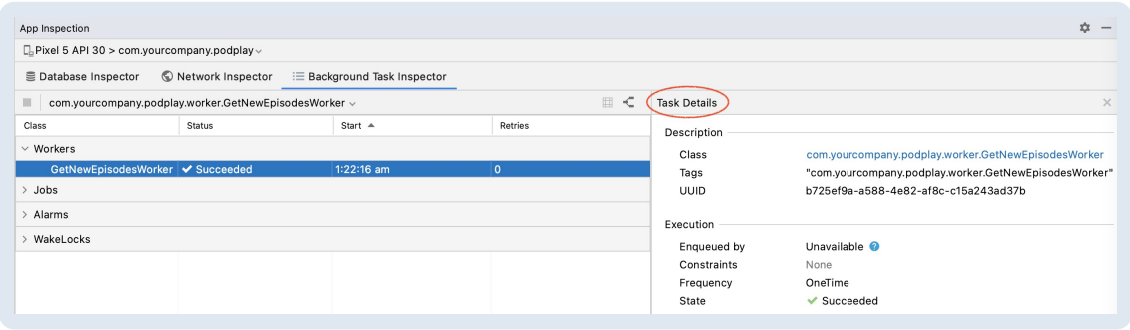
dropdown.



> **Note**: `WorkManager` adds a tag in: **[Your Package Name].worker.[Worker Class Name]** format to each worker if the tag isn't explicitly set while enqueuing the work.

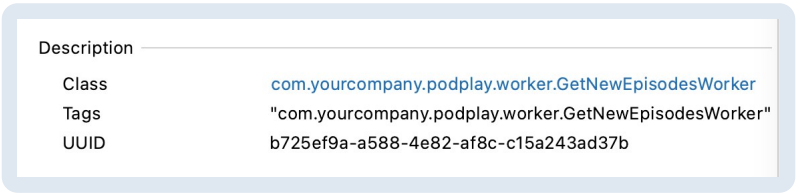
Anatomy of a Worker

Now that you've filtered `GetNewEpisodesWorker` from the table, click **the row** to reveal details about the work. The **Task Details** panel will open next to the row.



The panel breaks down the information into four major segments:

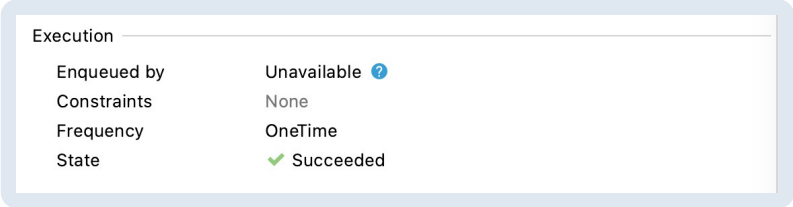
1. **Description** - This section explains the worker class itself.



The fields describe the following:

- **Class:** Displays the worker class name and the fully-qualified package name.
- **Tags:** The default tag assigned by `WorkManager` or any additional tags specified by the developer to identify this worker while creating it.
- **UUID:** An unique identifier provided by `WorkManager` to each worker.

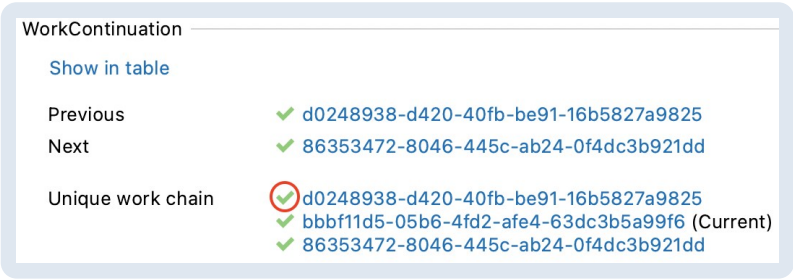
2. **Execution** - Presents the execution criteria of the worker.



This displays four important bits of information:

- **Enqueued By:** Shows the class name which created and enqueued this worker. The class name is only shown if that worker is enqueued *after* opening the table; otherwise, it shows as "Unavailable".
- **Constraints:** Displays details if the worker is running under any work Constraints. Constraints are one or more rules that must be met before executing the work as pre-conditions, for example, internet connectivity. If no constraints are set, then it simply displays "None".
- **Frequency:** Indicator that represents how many times the worker's going to execute the work. It can be a OneTimeWorkRequest or PeriodicWorkRequest that's scheduled and repetitive.
- **State:** The work's current state, either it's Enqueued, Running, Blocked, Succeeded or Failed.

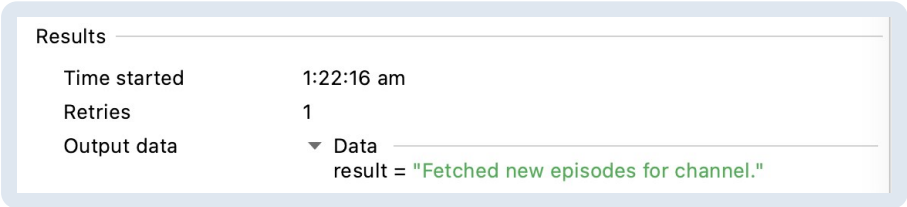
3. **WorkContinuation** - If the worker enqueued as part of any work-chain, this section would display its details.



Important fields to look at in this segment:

- **Previous:** UUID of the previous worker in the work chain, if any.
- **Next:** UUID of the next worker in this work chain (if available). It displays "None" if no worker is queued after this.
- **Unique work chain:** This is a quick overview of your work chain. It lists the UUIDs of all the workers in the sequence that they are executed. The worker marked as "Current" is the worker you're viewing details of right now. The tick icons beside each UUID mark that the task has been executed successfully.

4. **Results** - This section is for displaying the final results of executed work.

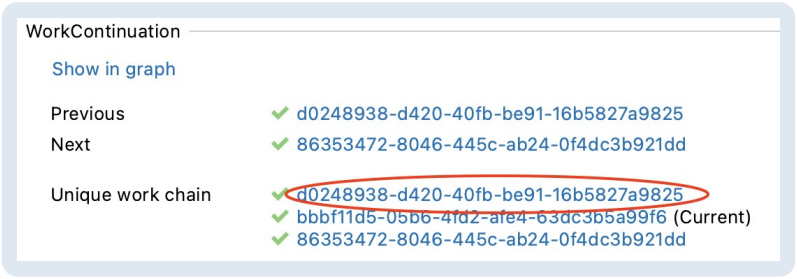


It consists of three specific pieces of data:

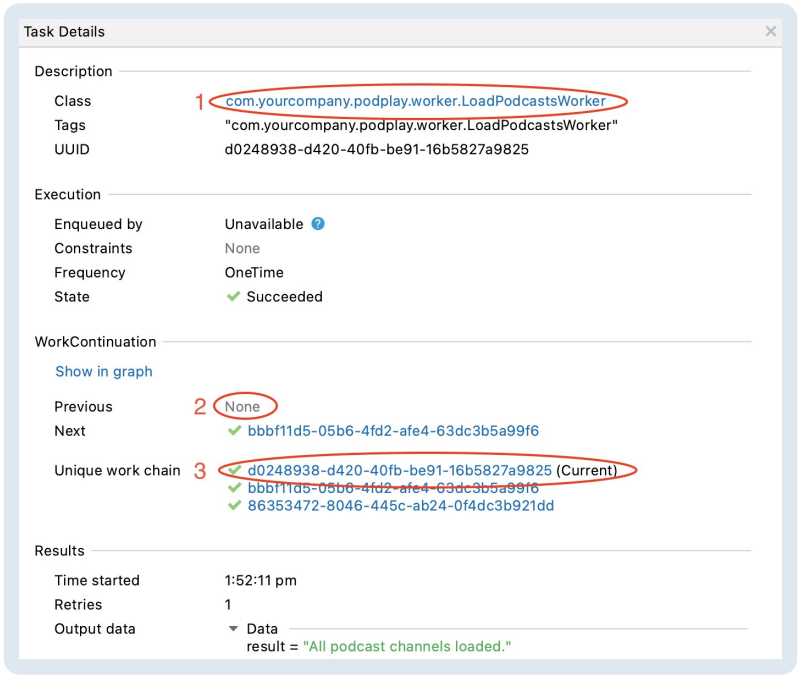
- **Time started:** The start time of the work execution.
- **Retries:** Shows the number of times the work has been executed, which is one, in this case. If work failed for some reason, the retry count would have increased.
- **Output data:** The returned result or data from `doWork()` in your worker class.

Navigating Through Works

The **WorkContinuation** section is interactive; You can easily navigate to any worker in the work chain simply by clicking on the worker's UUID.



Click the first **UUID** from the **Unique work chain** to preview its details. The **Task Details** panel changes, as you can see below:

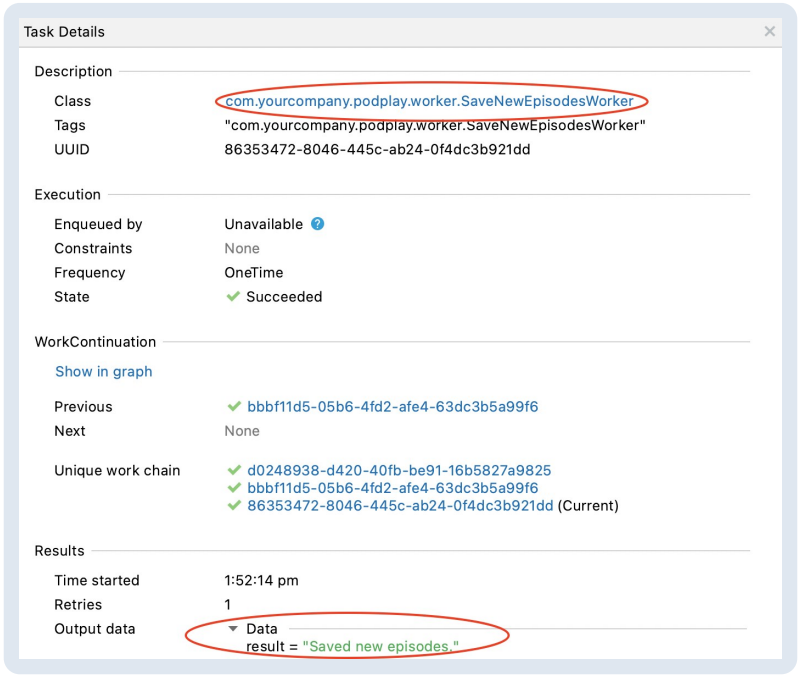


Notice the highlighted areas above. The changes are:

1. The class changed to the `LoadPodcastsWorker` path.
2. The previous work is “None”.
3. The system marks this first UUID in **Unique work chain** as “Current”.

Now the Task Details panel show for your first worker `LoadPodcastsWorker`, which means you’re pointing at the beginning of your work chain.

Next, click on the last UUID in **Unique work chain**; it will navigate you to the end of the work chain:

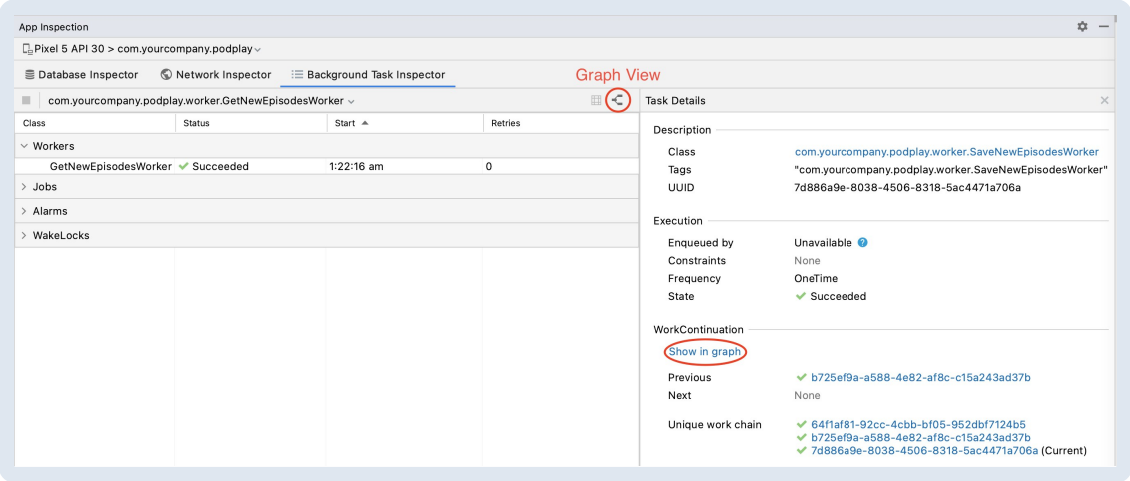


Look at the Task Details panel again; it shows the worker class is

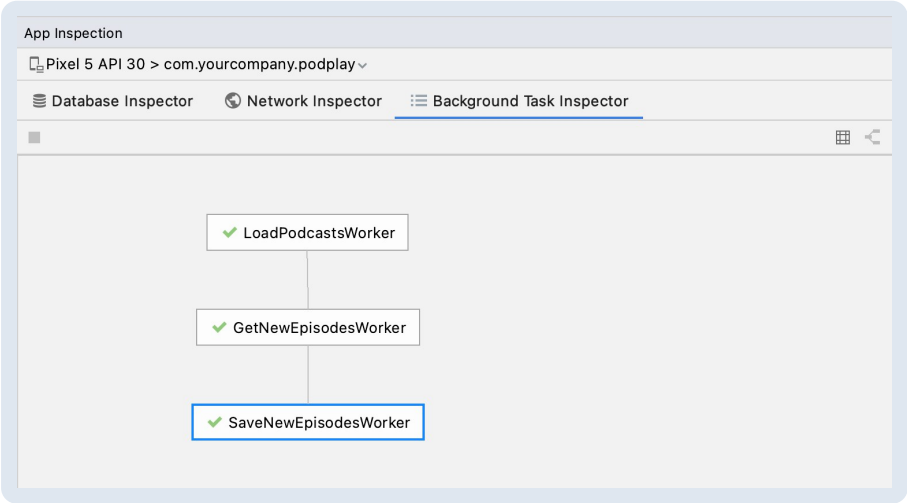
`SaveNewEpisodesWorker` and outputs “Saved new episodes” in the **Output data** section, meaning the end of the work chain. As in the previous example, the **Current** mark moved next to the selected UUID.

The Flow Diagram

The table can generate a nice graph of workers for better visualization. Click any of the highlighted areas below to bring out the **Graph View**:



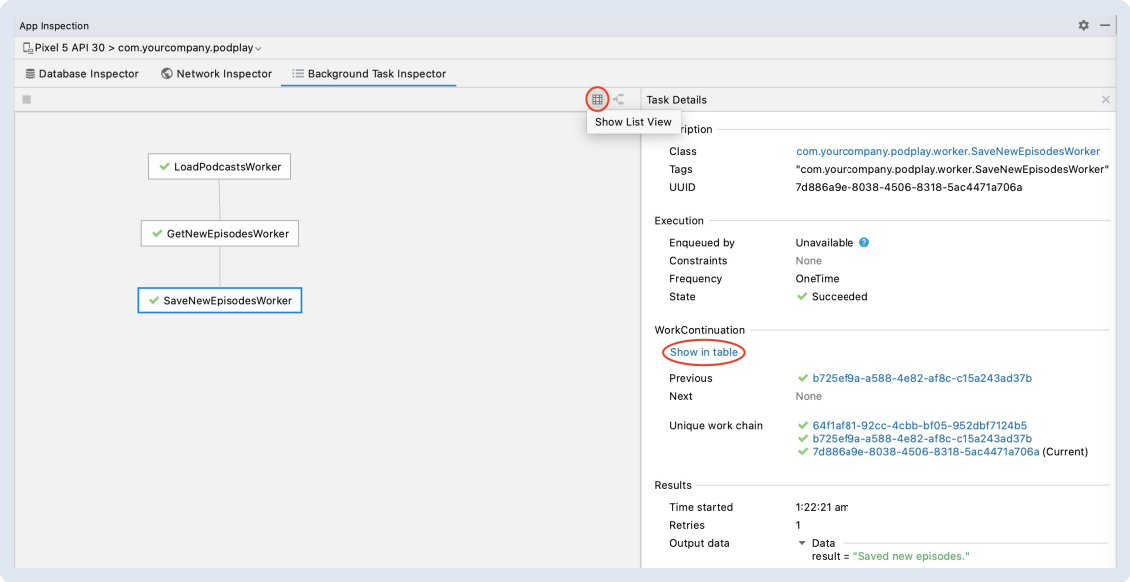
The graph presents each work from the execution flow, maintaining the exact order. So, the graphical representation of your work chain looks like this:



It’s visible from the graph that there are three workers in the work chain:

1. `LoadPodcastsWorker` executed first and loaded all podcast channels.
2. Then `GetNewEpisodesWorker` fetched new episodes for each channel.
3. Finally, `SaveNewEpisodesWorker` saved new episode information and finished the operation.

Click on of the highlighted points below to easily switch back to the table view:



Practical Example: A Simplified Work-Chain

In a real-world app, you might need to combine all the tasks of loading podcasts, fetching episodes, and saving updates in a single worker. The worker should run periodically in the background and only notify the user if new episodes have been added.

`EpisodeUpdateWorker` in PodPlay does all the heavy lifting mentioned above.

Open **EpisodeUpdateWorker.kt** and look into `doWork()`:

```
override suspend fun doWork(): Result = coroutineScope {
  val job = async {
    // 1
    val db = PodPlayDatabase.getInstance(applicationContext, this)
    val repo = PodcastRepo(RssFeedService.instance,
db.podcastDao())
    // 2
    val podcastUpdates = repo.updatePodcastEpisodes()
    // 3
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
      createNotificationChannel()
    }
    for (podcastUpdate in podcastUpdates) {
      displayNotification(podcastUpdate)
    }
  }
  job.await()
  Result.success()
}
```

What the `job` doing here is:

1. Loads `PodcastRepo`, the repository where all your bookmarked podcast channels are stored.
2. Calls `updatePodcastEpisodes()` that fetches the latest episode list from the channels and returns updates as a list in `podcastUpdates`.
3. It displays a notification to the user for each item in `podcastUpdates`.

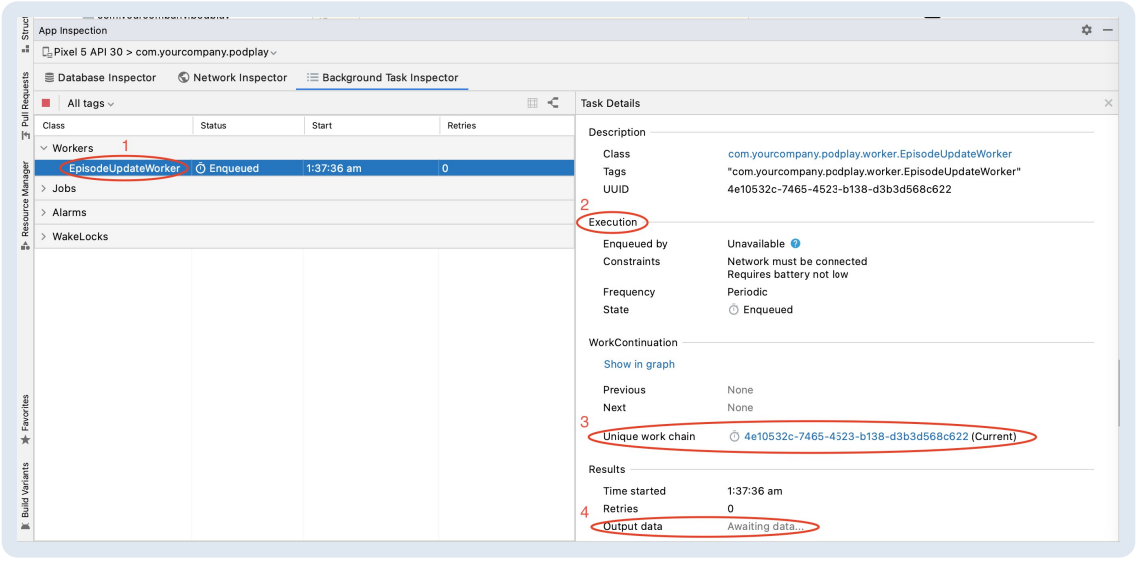
Your next step is to schedule this `job` using `WorkManager`. To do so, open **PodcastActivity.kt** and replace `scheduleJobs()` as follows:

```
private fun scheduleJobs() {
  // 1
  val constraints: Constraints = Constraints.Builder().apply {
    setRequiredNetworkType(NetworkType.CONNECTED)
    setRequiresBatteryNotLow(true)
  }.build()
  // 2
  val request = PeriodicWorkRequestBuilder<EpisodeUpdateWorker>(
      repeatInterval = 1,
      repeatIntervalTimeUnit = TimeUnit.HOURS)
      .setConstraints(constraints)
      .build()
  // 3

WorkManager.getInstance(this).enqueueUniquePeriodicWork(TAG_EPISODE
_UPDATE_JOB,
      ExistingPeriodicWorkPolicy.REPLACE, request)
}
```

The above code:

1. Defines a work constraint to run the job only when the device is connected to the internet and the battery isn't low. This ensures optimum resource consumption when the worker is running or prevents the worker from running if the conditions aren't met.
2. Creates a PeriodicWorkRequest that's scheduled to execute once every hour if the work constraints are satisfied.
3. Enqueues the periodic request to `WorkManager` with a tag. `ExistingPeriodicWorkPolicy.REPLACE` makes sure if there's an existing work request with the same tag, this will replace the existing one.

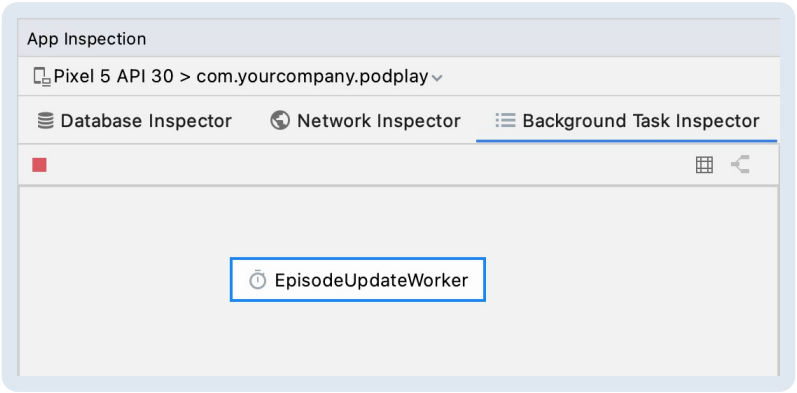
Now, build and run the app and observe the work from **Background Task Inspector**; you'll see some interesting developments!



The changes you see on the highlighted points are listed below:

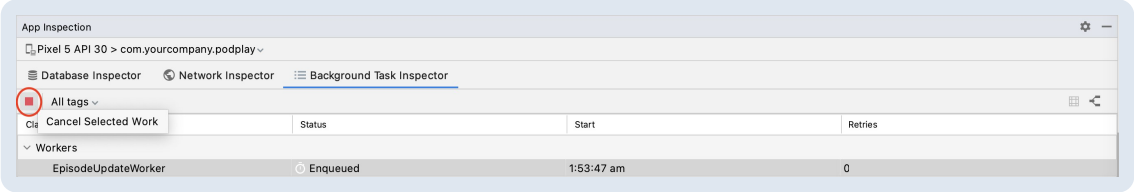
1. `EpisodeUpdateWorker` is now enqueued in the table of workers.
2. The **Execution** section under the Task Details panel now shows your defined constraints for the worker. It also displays that the work will execute at a periodic frequency, not one at a time.
3. The **Unique work chain** indicates that `EpisodeUpdateWorker` is the only item in the work queue displaying a single UUID.
4. The **Results** section is showing "Awaiting data..." for output because the work is scheduled to be executed in the future.

Switch to the **Graph View** that'll display a single item now:

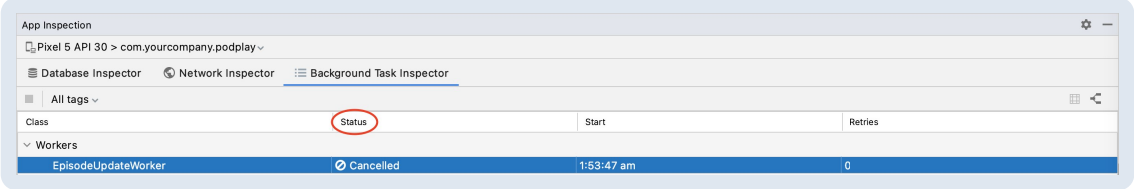


Canceling a Worker

Sometimes you may want to cancel a worker which is scheduled to run in future. It's easy to stop a currently running or enqueued worker using **Background Task Inspector**. Select `EpisodeUpdateWorker` and click the highlighted icon from the toolbar:



This will suspend any pending or scheduled work; you'll see the change immediately in the **Status** column.



Next, you'll learn how to monitor network activity with Network Profiler while you add new episodes in PodPlay. Good luck on your next quest!

Key Points

- You can easily inspect `WorkManager` workers using Background Task Inspector.
- Background Task Inspector contains a table with all workers.
- You can find a specific worker using tags.
- The Task Details panel reveals the current state, execution flow, work continuation and results of background work.
- You can see a detailed work chain graph using Graph View.

Where to Go From Here?

You've now mastered inspecting background tasks in Android! To learn how to utilize Background Task Inspector more, check out View and inspect Jobs, Alarms, and Wakelocks.

But don't stop there; take a look at our tutorials related to the background processing:

- Android Background Processing
- Scheduling Tasks With Android WorkManager

9 Profile CPU Activity

Written by Zac Lippard

Efficiently using the CPU is critical when it comes to your Android app. It determines the difference between jittery and smooth UI and significantly impacts a device's battery life.

It's important to keep in mind your app's impact on resources. Your app should run smoothly and feel fluid throughout. A choppy app won't bring delight to your end-user.

Your app should also be efficient and not have unfettered access to system resources. Extensive use of system resources causes other applications to delay their operations. This also drains the battery and could lower the app's overall performance.

This chapter covers the Android Studio **CPU Profiler** and how you can use this tool to track **CPU usage**.

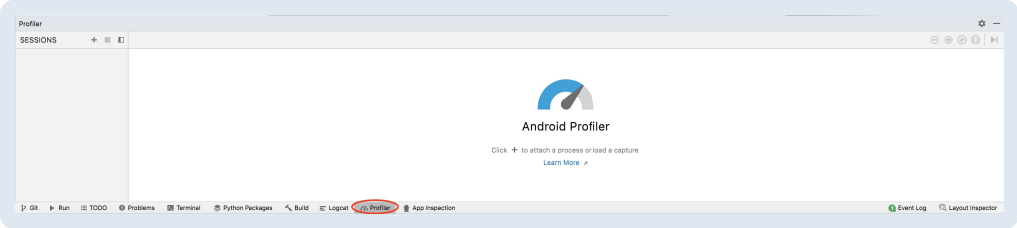
You'll learn:

- All about CPU traces including **system traces** and **method/function traces**.
- Recording traces.
- Import and export traces.
- Inspect traces using the many different charting formats available in the CPU Profiler.
- Use the third-party tool **Perfetto** to inspect system traces.

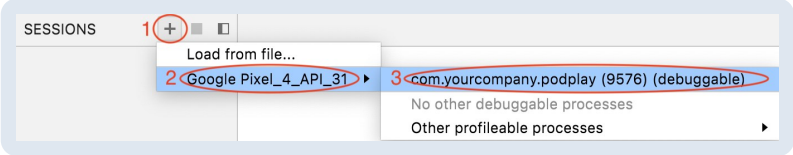
The CPU Profiler

Android Studio provides the **CPU Profiler** as a tool to measure CPU usage in real-time as well as to monitor active threads of an Android process. If you're noticing a performance issue with your app, the Profiler is there to help!

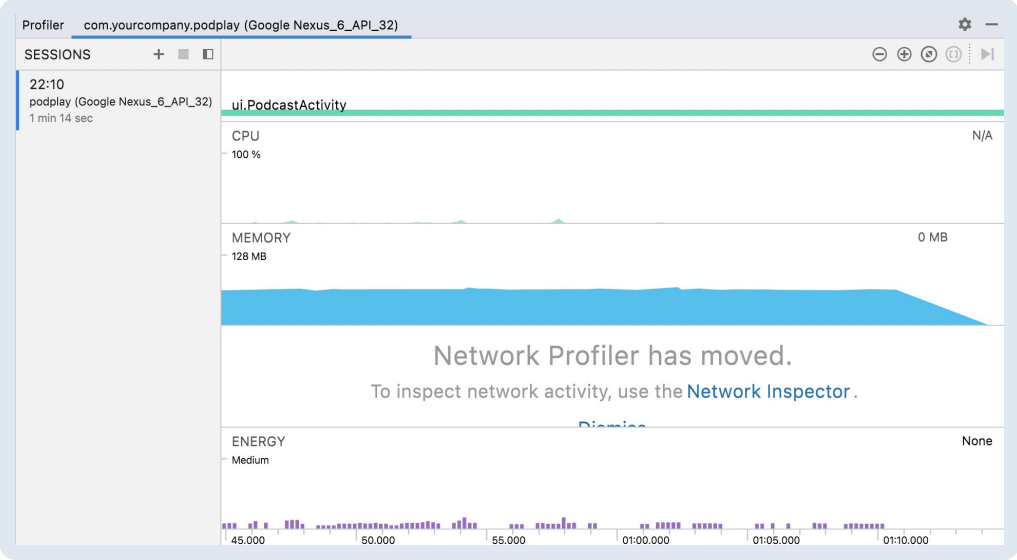
To start, open the **Podplay** starter project in Android Studio and run the application. Once the app is running, you can connect the Profiler to your app's process. To do so, click **View ▸ Tool Windows ▸ Profiler**, or simply click the **Profiler** tab in the bottom toolbar.



In the Profiler window, you'll notice there's not really much happening there yet. You now need to connect the Profiler to your running Podplay app process.



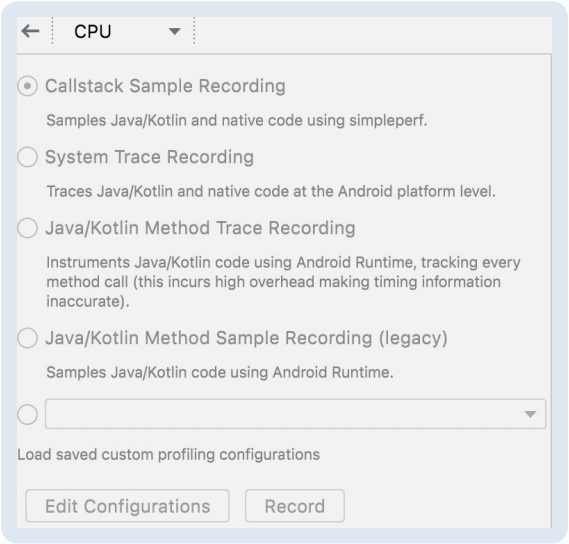
1. Click the + icon in the top-left of the Profiler window in the **Sessions** toolbar.
2. Select your device.
3. Choose the Podplay app process to connect to.



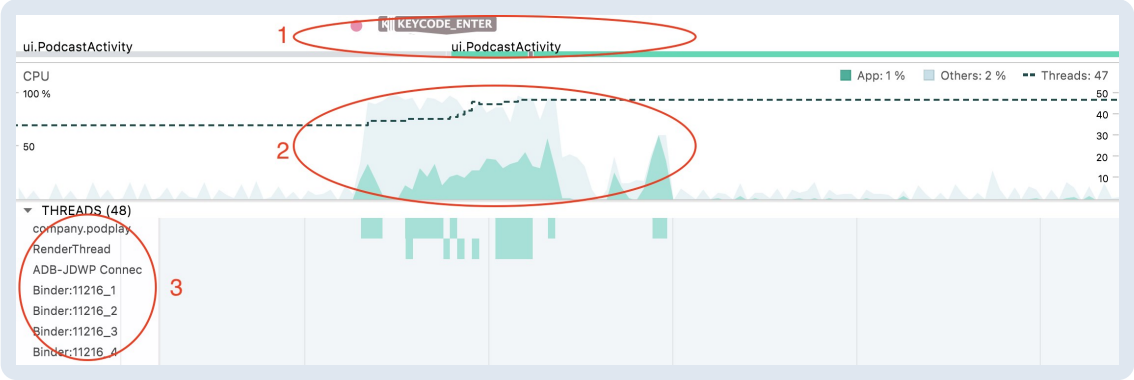
You'll notice a new entry on the left in the **Sessions** column. On the right, the Profiler has started collecting data. The session will remain until you close Android Studio. There are sections for CPU, Memory, Networking, and Energy. For the purposes of this chapter, you'll focus on the CPU section.

Double-click the **CPU** section, and the Profiler will update to show the CPU panel.

To the left of the CPU panel is a section to create trace recordings and configurations.



On the right-hand side, there's a series of timelines.



Each timeline section provides unique details about how your app is interacting with the CPU:

1. **Event timeline**: UI rendering and interaction data shows up here. Activity and fragment lifecycle calls will appear in this timeline. Any user interactions such as button taps or text input will also display in this section.
2. **CPU timeline**: This timeline displays the CPU utilization as a percentage. The timeline will also show CPU utilization belonging to other processes to compare your app against.
3. **Thread activity timeline**: Active threads that are running as part of the Podplay application process. A **green** section indicates the thread is active and running, or at least in a runnable state. **Yellow** represents that the thread is active but waiting for an IO operation. **Gray** shows threads that are sleeping and not consuming any CPU.

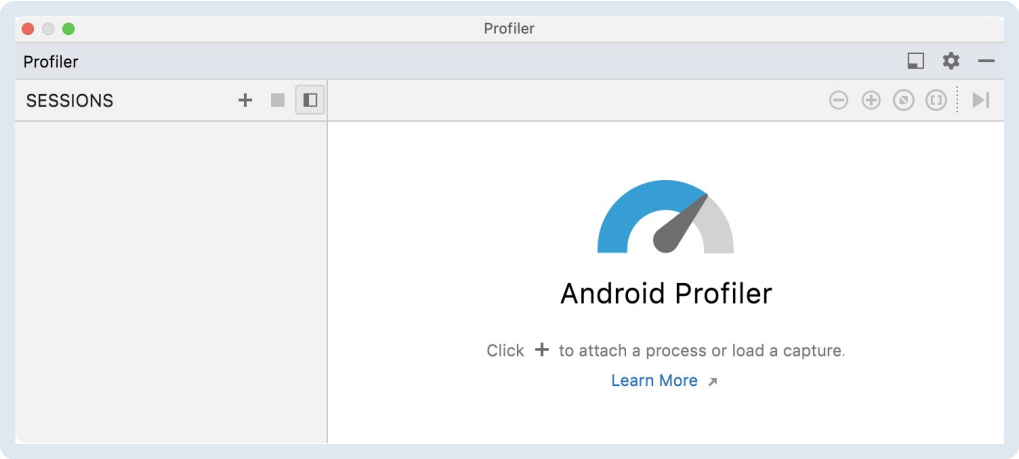
Standalone Profiler

Don't want to run the entire Android Studio IDE just to launch the Profiler? No worries! There's a way to launch just the Profiler as a standalone tool.

First, ensure Android Studio isn't running the profiler, or close Android Studio altogether. Next, open a terminal window to the `bin` folder under the Android Studio installation:

- **Windows/Linux**: `<studio-installation-folder>/bin`
- **macOS**: `<studio-installation-folder>/Contents/bin`

Then, depending on your OS, run `profiler.exe` or `profiler.sh`.



A new Android Studio profiler dialog will appear where you can start new sessions, same as listed above via the Profiler window.

One of the critical components of the CPU Profiler is the ability to record traces. However, before diving into recording and configuring traces, you'll learn what traces are.

Traces

A trace is a set of data that details how a process and its threads interact with the CPU and other resources. Typically the trace data is extremely thorough and covers method/function calls, rendering data, lifecycle calls on the Activity or Fragment and user input. All of these combined helps to paint a picture of how your app performs and what resources it's using.

There are two types of traces:

- **System Traces**: Details how your app as a whole interacts with resources on the given device.
- **Method/Function Traces**: Provides information about your app's Java methods and C/C++ functions. CPU utilization allocated to the method is also recorded in the trace. These types of traces also show a chain of caller and callee methods, very similar to a stack trace.

You can use trace data to see where your app may be utilizing more resources than necessary.

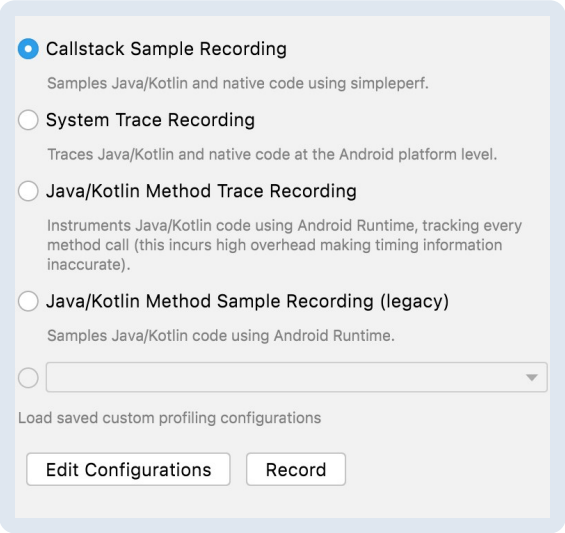
Now that you know what traces are, it's time to record one!

Recording Traces

Run the Podplay project in Android Studio. Start a new CPU Profiler session by clicking on the **Profiler** window tab at the bottom of Android Studio. Click the **+** icon and select the device and Podplay process to launch the new session. Now, click the **CPU** timeline to open the CPU Panel.

Recording Configurations

As mentioned earlier in the chapter, the left column shows a number of report configurations.



The configurations will capture different data depending on your need:

1. **Callstack Sample Recording**: This will capture Java/Kotlin method and C/C++ function call stack information using `simpleperf`. For native code sampling, you must be recording against a device running Android 8.0+ (API 26+). `simpleperf` is a command-line tool that you can use to run trace reports outside of Android Studio and the CPU Profiler itself. The tool has several options you can configure to this recording configuration. For more details, read the Android documentation.
2. **System Trace Recording**: Provides fine-grained details about the system resources, like CPU core utilization and memory usage. You'll need an Android 7.0+ (API 24+) device to record with this configuration.
3. **Java/Kotlin Method Trace Recording**: This configuration adds instrumentation to your app at runtime. The instrumentation adds

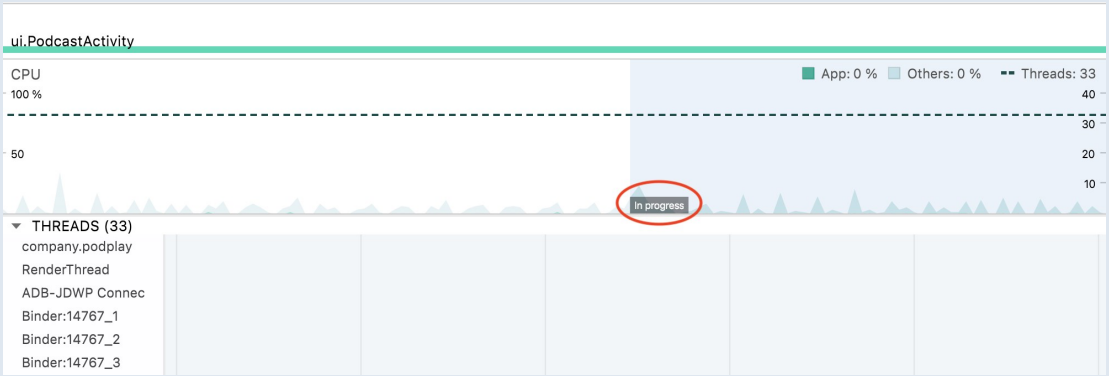
timestamps to the beginning and end of each method in the call stack. These timestamps are then used and aggregated to calculate app timing as well as CPU utilization. Note that this configuration adds significant overhead due to the required instrumentation. For short-lived methods, this configuration could result in inaccurate time data. If your app makes a large number of method calls throughout the recording, it could cause the Profiler to exceed its maximum file size limit.

4. **Java/Kotlin Method Sample Recording (legacy)**: Similar to the **Callstack Sample Recording** option but uses the legacy `systrace` tool. This works on devices using Android 4.3+ (API 18+).

Recording a Trace

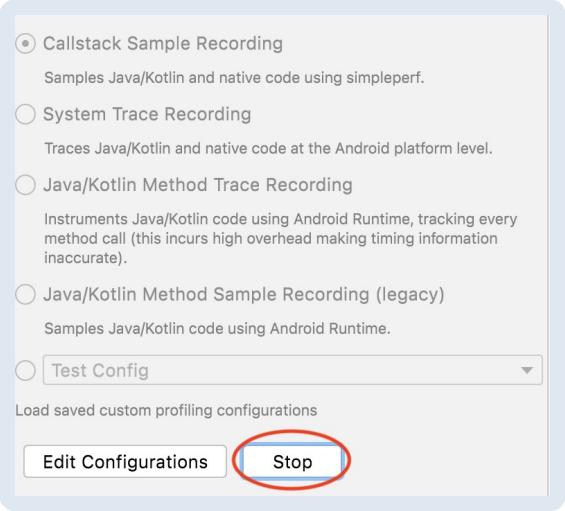
To record a trace from the CPU Panel, click **Record**.

You'll notice that once the recording starts, the CPU timeline background changes color, and there's an "In progress" tooltip in the timeline at the start of the recording.

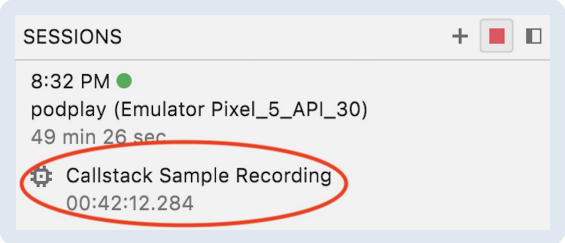


Note: Depending on your workstation, trace recordings can be resource intensive, especially if you're using an emulator or the method trace recording configuration. If you notice latency while recording, try switching over to an actual device to reduce some of the overhead on your workstation.

Now that you've collected some data in the recording, click **Stop**.



Notice that the active session adds the new recording:

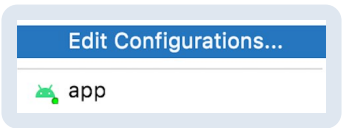


The recording entry includes the configuration type and a timestamp of when the recording occurred during the profiler session.

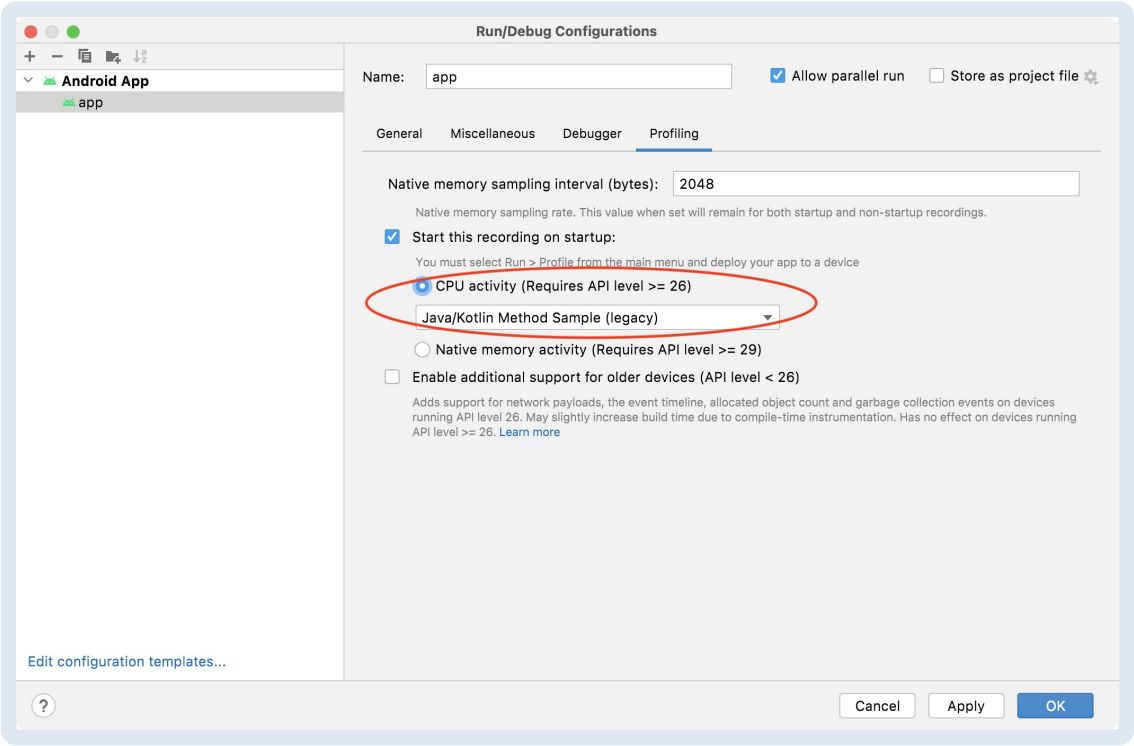
Recording at App Startup

You can create trace recordings during the app startup as well. This is useful for monitoring resources and code during the launch of the `Application` and initial `Activity` classes where you might not be quick enough to attach the Profiler manually.

First, click **Select Run/Debug Configuration** at the top of Android Studio. Click **Edit Configurations...**.



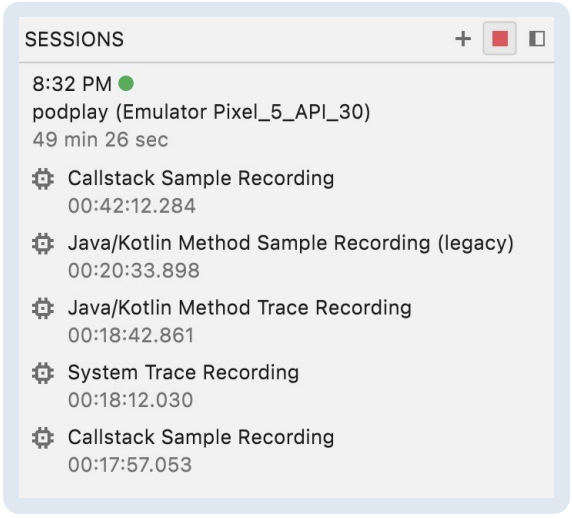
Click the **Profiling** tab. Here you can check the **Start this recording on startup** option. Select the **CPU activity** option and choose one of the record configurations.



Click **OK** to save the run configuration and close the dialog. Then, in Android Studio, choose the **Run ▸ Profile...** menu option, or click on the **Profile 'app'** icon:

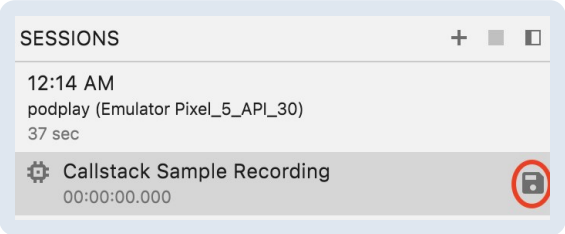
Notice that the Profiler immediately starts when the app launches. A new session will appear in the left column along with the recorded trace during the app's run.

Multiple recordings all appear under the associated session:



Exporting a Recording

Now that you've acquired a number of recorded traces, it's time to export them. Hover over one of the recordings, and you'll notice a save icon that you can click.



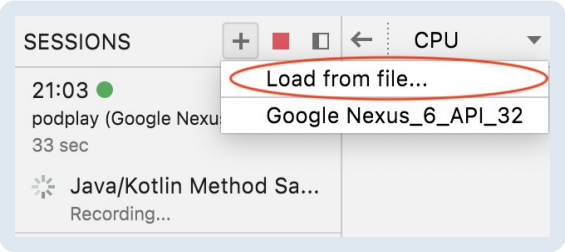
Click the save icon, then in the new dialog, click **Save**. Your trace recording is now saved in a `.trace` file.

Importing a Recording

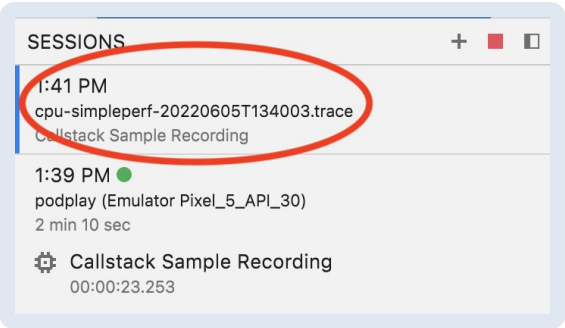
You can also import recordings to Android Studio. This is useful if you want to share and access a recording from another workstation or with other developers.

To import a recording, click the **Profiler** tab to open the profiler window.

Next, click the + icon and choose **Load from file...**.



Select the recording that you exported in the previous section. A new session will appear with the imported trace recording:



The trace filename and the associated recording configuration type appear in the session entry.

Congratulations on successfully importing a recording! Now that you've gathered the trace data, it's time to inspect it.

Inspecting Method Traces

Within the CPU Profiler, there are a number of ways to be able to inspect the trace data. You'll learn about the different charting approaches in this section and the benefits of using each one.

To start inspecting a trace, you'll need to record a new one.

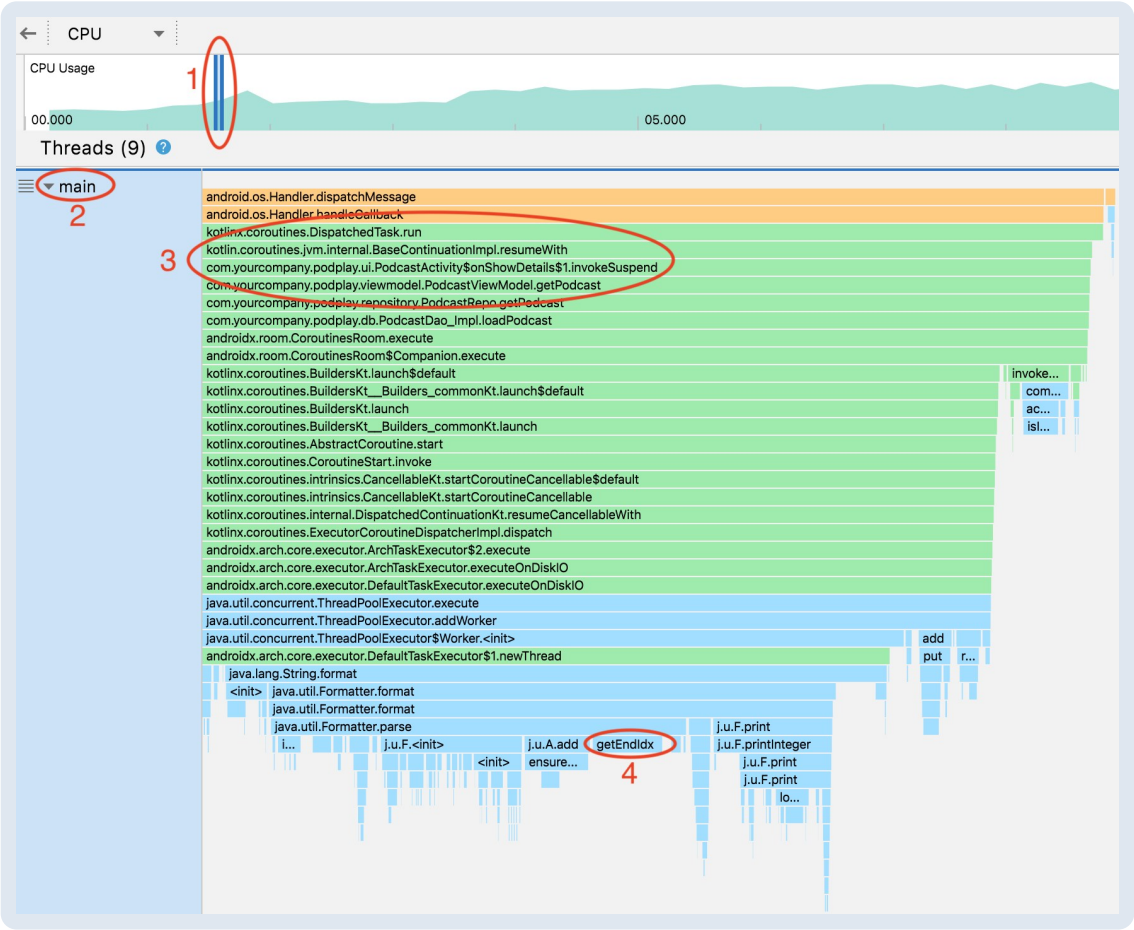
Begin by running Podplay in Android Studio. Once the recording has started, tap **Search**. Search for "android". Once the podcasts load, it's time to start the profiler session.

Start a new profiler session and record a new trace using the **Java/Kotlin Method Trace Recording** configuration. Once the recording starts, go back to the app and tap on a **podcast**. When the podcast details page loads, go back to the profiler window in Android Studio and click **Stop** to end the recording.

Now that you have a trace, it's time to use the different charts to inspect it!

Call Chart

The first charting option is the **Call Chart**. It appears under the **Threads** section of the trace. This chart provides a visual aid of the method and function calls made during the timeframe of the recording.



1. The CPU Usage timeline shows an overall CPU utilization as a timeline. You can drag the vertical blue start and end lines to focus on a specific portion of the timeline. Doing so adjusts the associated data and will update the details in the other places like the **Threads** section.
2. Select the **main** thread and expand it by clicking the arrow. Notice that the call chart updates and reflects all the methods and native functions called by the app during the time window you selected. There are three different colors used to represent the different method calls in the chart:

- **Orange** indicates calls by the app to system APIs.
- **Green** are the app's own methods.
- **Blue** are third-party API calls.

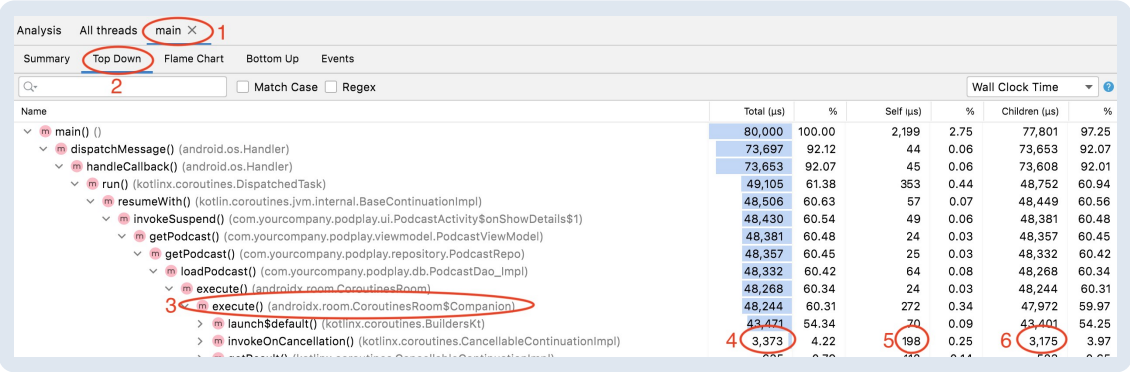
3. The method calls in the Threads section display from top to bottom, where the topmost method is the first call, and the subsequent method below is the next call in the list. In this example, there are system calls at the top that launch a coroutine that eventually fires `PodcastActivity$onShowDetails$1.invokeSuspend()`. This method then calls the next method in the list `PodcastViewModel.getPodcast()`. The call list continues down from there.
4. Notice how the method lengths get shorter the lower down the list you go?

You can think of methods below a certain method in the list as "children". Each child runs as long or as short as the parent. In the example above, the child method `getEndIdx()` runs for a fraction of the time it's parent caller `Formatter.parse()` ran. These methods are all descendants of `PodcastViewModel.getPodcast()`.

The call chart in the Threads section is extremely valuable for determining what methods are taking longer than expected. You can drill down deeper to see if there are unnecessary calls to other methods or if there's a child method that's an underlying culprit to poor performance in the app.

Looking at Top Down Charts

The **Top Down Chart** provides similar details to the Call Chart mentioned earlier but with more fine-grained details about each method. To use this chart, switch to the **Analysis** section on the right-hand side.



Here's the breakdown:

1. Notice that the `main` thread is still selected.
2. Choose the **Top Down** tab to view the methods.
3. The call stack will display somewhat expanded. You can continue to expand them by clicking the arrows on each method to see its children. In the screenshot, `loadPodcast()` expands along with it's child `execute()`. The nested `execute()` then has a series of methods it calls. In the top down chart, child methods appear below their parents.
4. The nested `execute()` has a series of child methods within it. Notice that `execute()`, along with its ancestor methods, took 48,244 microseconds (~48 milliseconds) to run. The **Total (µs)** column represents the total amount of time the method took to execute code within itself and all the time its children took. The child `invokeOnCancellation()` took a total of 3,373 microseconds.
5. Regarding `invokeOnCancellation()` again, you'll notice in the **Self (µs)** column that 198 microseconds were allocated to the method itself for

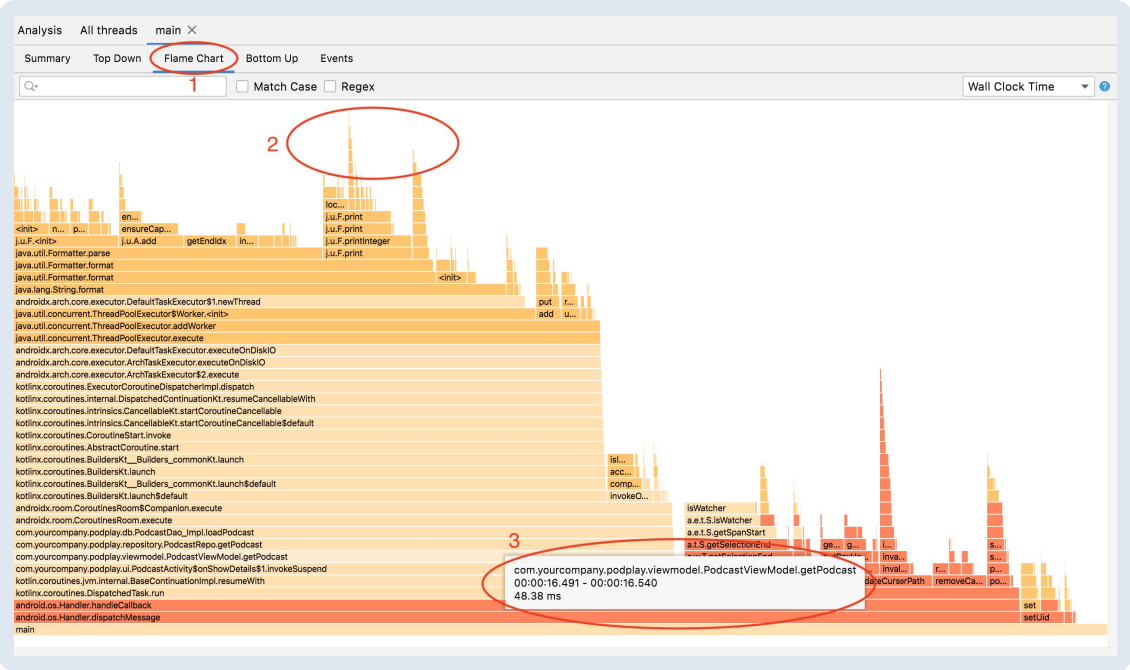
execution.

6. The remaining 3,175 microseconds were allocated to the children of `invokeOnCancellation()` via the **Children (µs)** column.

The granular breakdown of this chart provides you with a view of how long each method and child are taking. You can use this data to better understand how long methods are taking and begin to pinpoint why they run as long as they do.

Looking at Flame Charts

The **Flame Chart** option is an *inverted* Call Chart. This way, ancestor methods are at the bottom, and the descendants appear above them. This causes the visualization of the chart to appear like flames, hence the name.



Follow these steps:

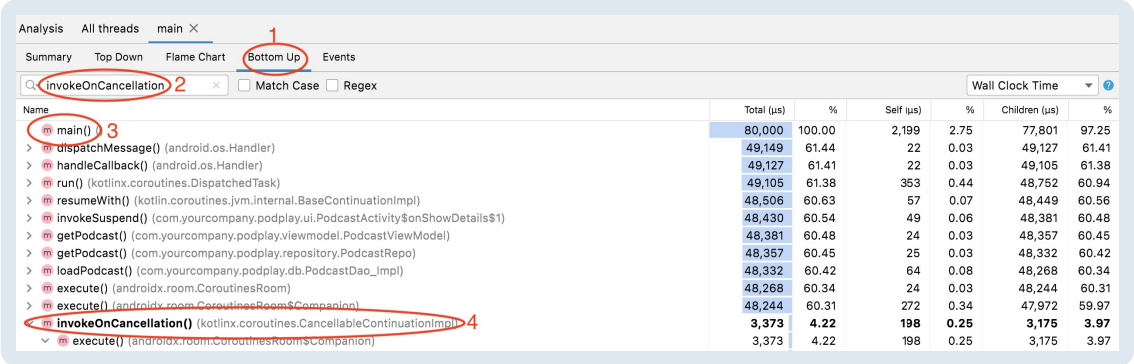
1. Click the **Flame Chart** tab at the top.
2. Note that the methods towards the top get smaller as these represent the child methods that will inevitably take less time to run than their parents, creating the appearance of flames burning across the screen.
3. You can hover your mouse over a method in the chart to get details about the method, such as the full method name, beginning and ending timestamps and the overall time taken in milliseconds. In this example, you can see that `PodcastViewModel.getPodcast()` took a total of 48 milliseconds to run.

The colors for the chart are slightly different compared to the Call Chart, but the same method types are represented here:

- **Dark orange** represents the third part API method calls.
- **Light orange** shows methods that belong to the app.
- **Red** displays system API calls.

Looking at Bottom Up Charts

As you may have guessed, the **Bottom Up Chart** is very similar to the Top Down Chart but *inverted*, in a way.



In this chart, you'll see all the methods called over the period of the recording at the top level. The difference though is that ancestor methods will have less associated with them.

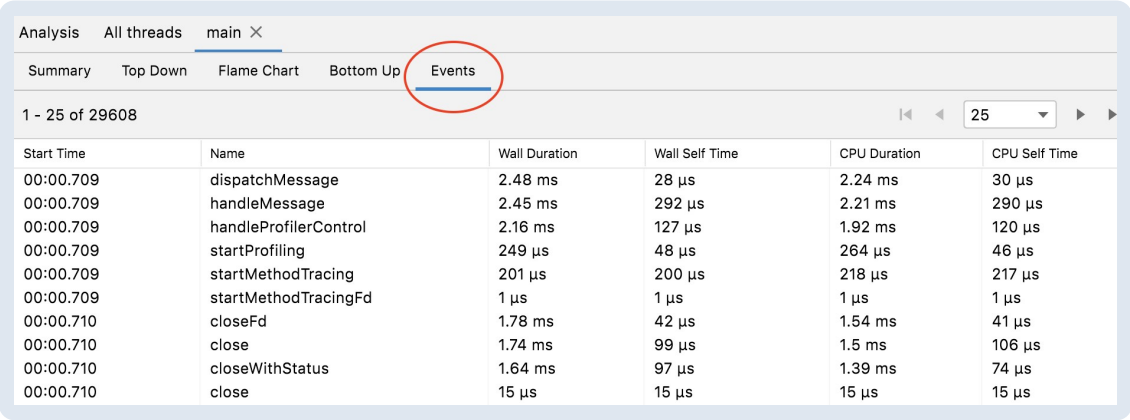
1. Click the **Bottom Up** tab to view this chart type.
2. Search for `invokeOnCancellation()` in the search box at the top.
3. Notice that `main()` can't expand. It's the root method and has no other parent tied to it.
4. Because of the search criteria, only the methods associated with `invokeOnCancellation()` will remain. That method itself will also be bold. You can expand this method to view `execute()` as its parent. You can keep expanding each ancestor method all the way back up to `main()`. Every method eventually routes back to `main()`.

The same columns are here in this chart as in the Top Down Chart to represent total/self/child time and percentage.

> **Note**: With any charts under the **Analysis** section, you can use the search box to bold methods that fit the search criteria. This is useful when you have many methods that you're trying to look through.

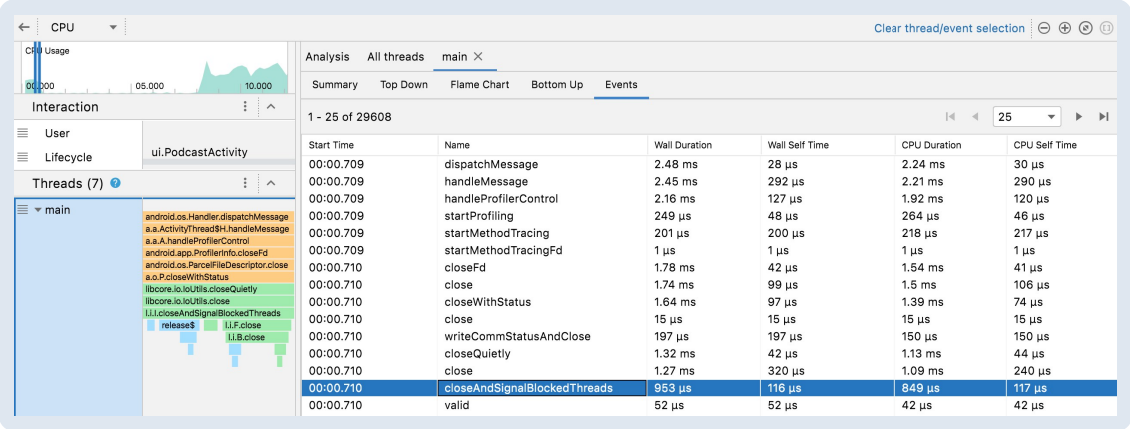
Events

The **Events** tab is useful to see all the calls for the thread selected in the **Threads** timeline.



Start Time	Name	Wall Duration	Wall Self Time	CPU Duration	CPU Self Time
00:00.709	dispatchMessage	2.48 ms	28 µs	2.24 ms	30 µs
00:00.709	handleMessage	2.45 ms	292 µs	2.21 ms	290 µs
00:00.709	handleProfilerControl	2.16 ms	127 µs	1.92 ms	120 µs
00:00.709	startProfiling	249 µs	48 µs	264 µs	46 µs
00:00.709	startMethodTracing	201 µs	200 µs	218 µs	217 µs
00:00.709	startMethodTracingFd	1 µs	1 µs	1 µs	1 µs
00:00.710	closeFd	1.78 ms	42 µs	1.54 ms	41 µs
00:00.710	close	1.74 ms	99 µs	1.5 ms	106 µs
00:00.710	closeWithStatus	1.64 ms	97 µs	1.39 ms	74 µs
00:00.710	close	15 µs	15 µs	15 µs	15 µs

Click an event to update the Call Chart in the Threads timeline to reflect the associated methods tied to the event.



Start Time	Name	Wall Duration	Wall Self Time	CPU Duration	CPU Self Time
00:00.709	dispatchMessage	2.48 ms	28 µs	2.24 ms	30 µs
00:00.709	handleMessage	2.45 ms	292 µs	2.21 ms	290 µs
00:00.709	handleProfilerControl	2.16 ms	127 µs	1.92 ms	120 µs
00:00.709	startProfiling	249 µs	48 µs	264 µs	46 µs
00:00.709	startMethodTracing	201 µs	200 µs	218 µs	217 µs
00:00.709	startMethodTracingFd	1 µs	1 µs	1 µs	1 µs
00:00.710	closeFd	1.78 ms	42 µs	1.54 ms	41 µs
00:00.710	close	1.74 ms	99 µs	1.5 ms	106 µs
00:00.710	closeWithStatus	1.64 ms	97 µs	1.39 ms	74 µs
00:00.710	close	15 µs	15 µs	15 µs	15 µs
00:00.710	writeCommStatusAndClose	197 µs	197 µs	150 µs	150 µs
00:00.710	closeQuietly	1.32 ms	42 µs	1.13 ms	44 µs
00:00.710	close	1.27 ms	320 µs	1.09 ms	240 µs
00:00.710	closeAndSignalBlockedThreads	953 µs	116 µs	849 µs	117 µs
00:00.710	valid	52 µs	52 µs	42 µs	42 µs

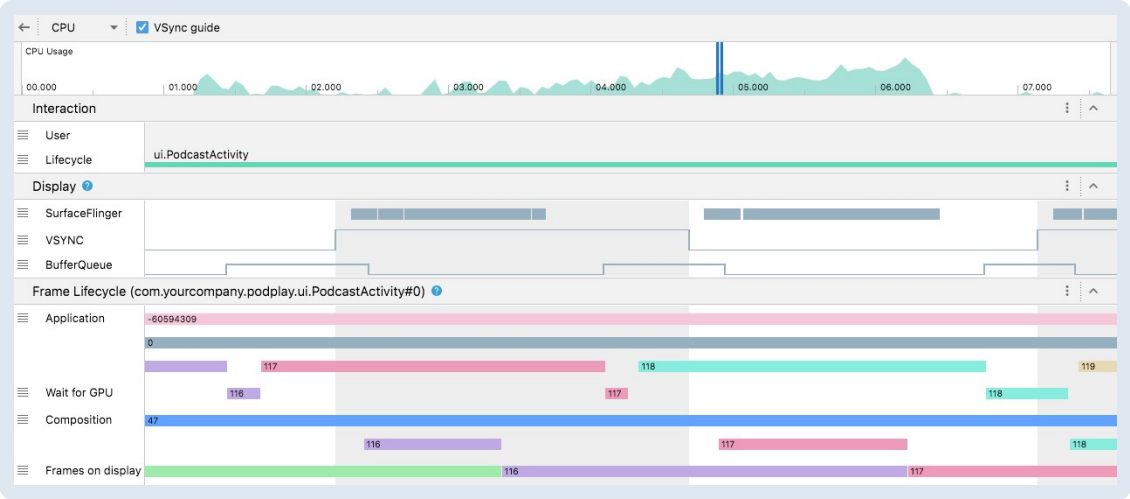
You've taken a deeper dive into method and function tracing, but now it's time to see the power behind system traces and what insights they provide into system resources.

Inspecting System Traces

As mentioned earlier, system traces provide fine-grained details on the system resources allocated to the app on a given device. This provides a different approach to understanding how performant your app is and how efficient it runs within the Android ecosystem.

Firstly, record a trace, but this time with the **System Trace Recording** configuration. You can follow the same steps as you did earlier by running the app and searching for "android" podcasts. Once the list of podcasts displays, stop the recording.

You'll now see the recording displayed in the profiler window.



UI Interaction

The **Interaction** timeline shows you what the user did during the time frame captured.

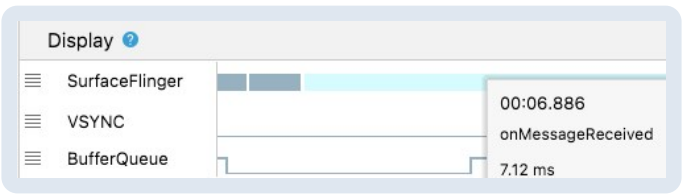
Action data recorded here are things like taps and text input. Lifecycle-related information for activities and fragments is also displayed here. You can hover over a specific interaction to get more details about the event's length.

The **Display** timeline shows information about drawn frames and what is set to draw next from the buffer.

The attributes for this timeline are:

- **Frames**: A frame is a UI rendering created by Android and displayed on the screen. The frame drawing measurement happens here. Long draw times display in red.
- **SurfaceFlinger**: This system process is responsible for sending buffer data to display.
- **VSYNC**: The signal indicates when to synchronize the display. This is a binary value of either 0 or 1.
- **BufferQueue**: Determines how many buffers are in the queue. These buffers then get processed by the SurfaceFlinger.

You can hover over the data in the timeline to see the values for each of these.



The **Frame Lifecycle** timeline shows the lifecycle of a particular "frame" that

displays to the user. `Activity` and `Fragment` instances build the screen associated with a given frame. In this example, the frame in the timeline associates with `PodcastActivity`.

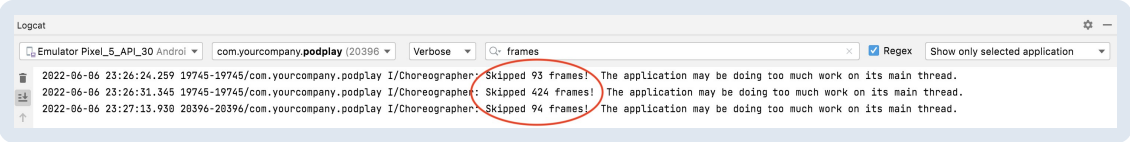
The attributes measured here are as follows:

- **Application**: The duration of the app drawing the frame to the screen.
- **Wait for GPU**: The time required for the GPU to complete the drawing.
- **Composition**: The composition time taken by the SurfaceFlinger.
- **Frames on display**: How long the frame displays on the screen.

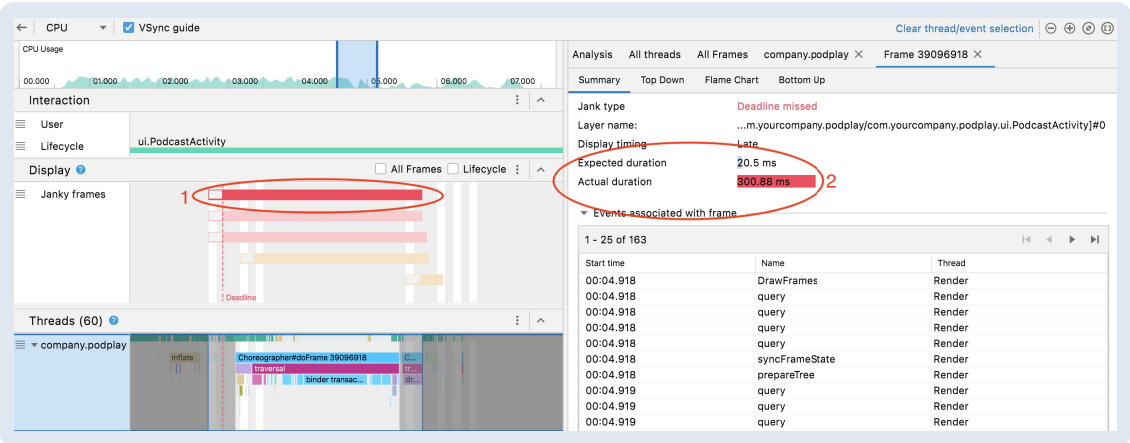
UI Jank

Jank refers to an unpleasant or unstable experience within an app. Specific to the user experience, UI jank is often referring to jittery UI that feels unnatural or jarring to the end-user.

Aside from visually inspecting the running app, the most common way to determine jank is to look for skipped frames in **Logcat**.



For devices running Android 12+, there is also a way to detect these janky frames from a system trace:

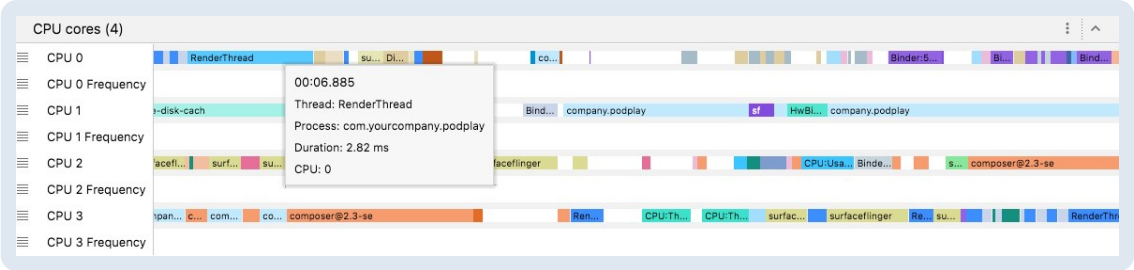


1. When there is jitter in the UI, the **Janky frames** section will appear in the **Display** timeline of a system trace.
2. Click a frame in the section to view details about the jank type as well as expected and actual durations it took for the frame to render.

CPU Cores

The **CPU Cores** timeline gives insight into how the work your app requires allocates to the device's CPU. This section lists each CPU core and frequency.

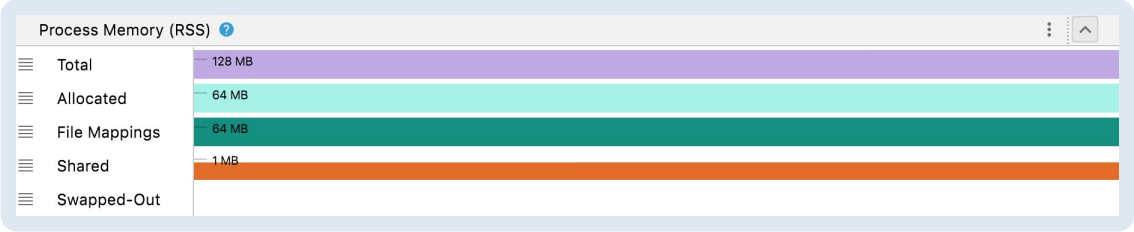
You can hover over a particular thread on one of the cores to get more information. Some of these details include how long each thread is taking and the associated process.



Process Memory (RSS)

The **Process Memory** timeline provides details about the app's memory footprint. This footprint, generally known as the **Resident Set Size** or **RSS**, describes how much memory the app uses in RAM.

This timeline breaks down the memory usage between shared and non-shared memory requirements. Non-shared memory is specifically allocated to the app for normal and file mapping purposes.



System Trace Formats

There are two primary formats for system traces on Android: **Perfetto** and **Systrace**.

In Android 10 and higher, system trace files save in the Perfetto format. This format includes more system data sources than Systrace does. You can record much longer traces with the Perfetto format. Along with the format, Perfetto is also the platform-wide tracing tool. Android, Linux and even Chrome all use this format. You can also use this tool via the command line with `perfetto`.

Systrace is a legacy system trace tool and format used in Android 4.3 and higher. Systrace files save as compressed text files. Thus, they cannot contain as much

data as their Perfetto counterpart. While Systrace is available on Android 10+ devices, Perfetto is the recommended format.

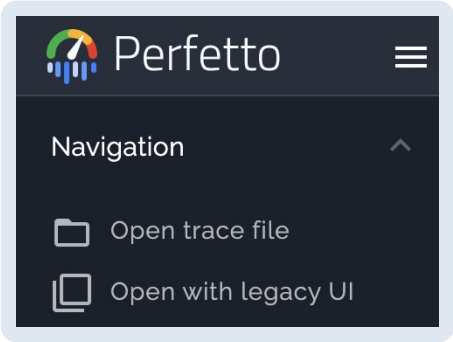
Inspection With Perfetto UI

A situation may arise where you have a system trace file but don't have Android Studio installed. Fear not, for there is a third-party alternative that you can use!

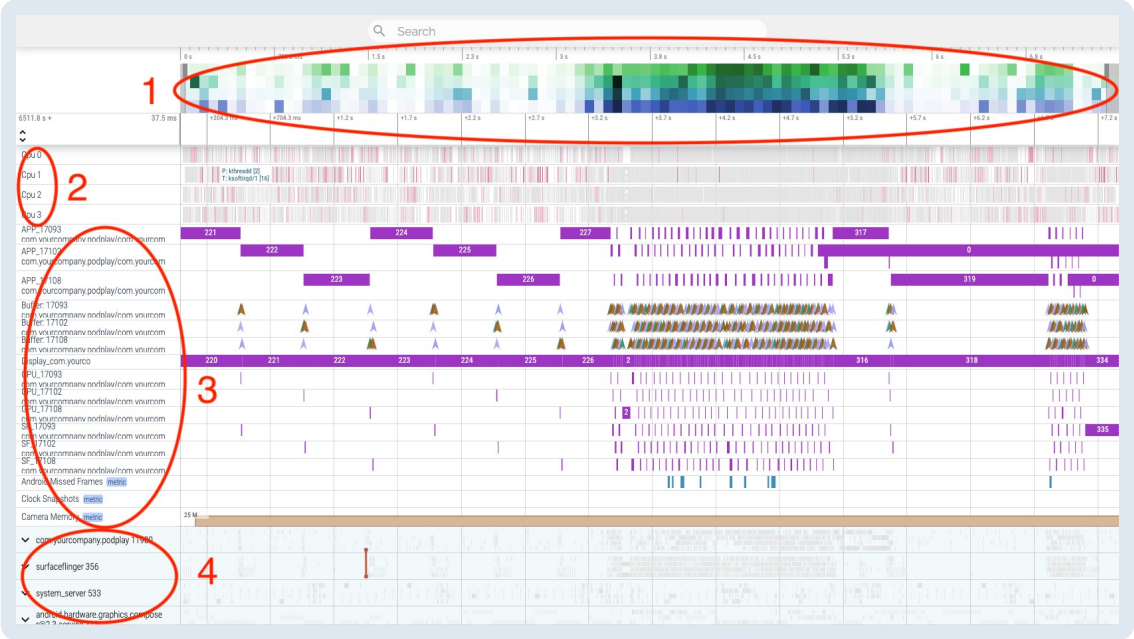
Perfetto UI is an online tool that you can use to parse and inspect Perfetto and Systrace formatted system traces.

To begin, take one of the system traces you recorded earlier in this chapter and go to Perfetto UI.

In the left column, click **Open trace file** and open either your Perfetto or Systrace formatted trace file.



Once you've selected your file, Perfetto UI will update and display the timeline.



1. Overall CPU activity at the top.
2. CPU core processing details.

3. Process breakdown by CPU activity.
4. Process and thread details.

Challenge

Remember the topic about UI jank? Well, it turns out there's some jankiness in this chapter's starter and final Podplay app as well. Open the **Logcat** window and run the Podplay app. Tap **Search** and search for "android". When the list of podcasts appears, tap any. Then, tap on a podcast episode to launch the player. You'll notice a delay before the loading spinner appears. During this time, you'll see logs stating `Skipped X frames!`.

Using the knowledge you gained from this chapter, record and inspect method or system traces to fix the UI jank and make it run smoothly!

> **Note**: The jank specific to this challenge is only in this chapter's starter and final Podplay projects. It isn't present in any of the other chapter's starter and final projects.

Key Points

- **System traces** measure performance on system resources and the UI.
- **Method/function traces** measure performance of targeted Java/Kotlin methods and C/C++ functions.
- Traces can be imported and exported with Android Studio.
- The CPU Profiler provides inspection on traces.
- You can analyze traces with a number of charting options provided by the CPU Profiler.
- **Perfetto UI** is an alternative to the CPU Profiler to analyze system traces.

Where to Go From Here?

You covered a lot of ground regarding CPU profiling. There's still so much more to explore regarding this topic!

Check out the Android developer CPU Profiler doc for more details on what the Android Studio CPU Profiler has to offer. Interested in learning more about the types of system tracing tools at your disposal? Check out this system tracing Android developer doc.

Now that you've learned about the CPU aspect of the Profiler, it's time to take a look at how you can use the Profiler for memory usage.

10 Profile Memory Usage

Written by Zahidur Rahman Faisal

Memory is a storage space in computers or mobile devices, just like the human brain! This is a place where data or programs are kept on a temporary or permanent basis to be processed.

Any smartphone app like PodPlay keeps occupying your device's memory as long as it's running in the foreground or background. It's important to ensure your app is functional with minimal memory usage and leaves enough room for the Android OS and other apps to operate correctly. The **Memory Profiler** is a tool built within Android Studio that helps you understand, analyze and optimize your app's memory usage.

In this chapter, you'll use the Memory Profiler and learn:

- The basics of **memory management** in Android.
- Tracking memory **allocations**.
- About **memory leaks** and how to prevent them.

Before moving any further, you might want to recall Android's memory management basics.

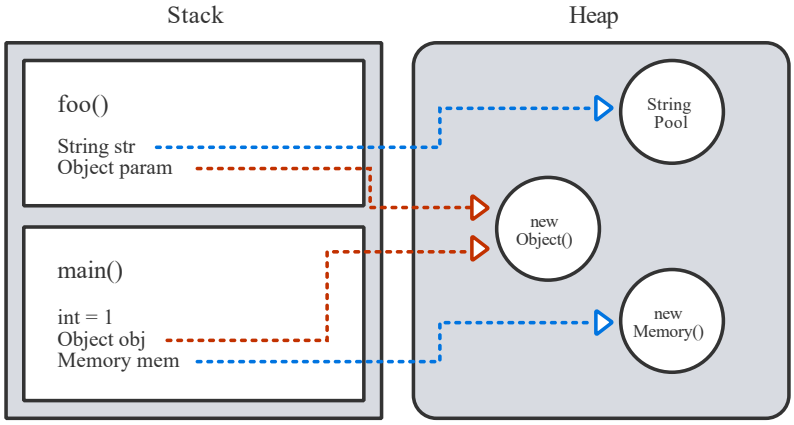
Android Memory Management 101

Android is a **managed-memory** environment. The Android Runtime (ART) and Dalvik Virtual Machine use memory paging and mapping techniques to manage memory for your Android device. Below are the key elements of Android memory management:

- **Memory Allocation**: This is the process of reserving memory for your app's objects and processes.
- **Stack Memory**: Android uses this for static memory allocation. Local variables, references to objects and primitive types are common examples of static memory. Each thread has its separate stack, organized in a LIFO order (last in, first out). The unified stack size on Android Runtime for both Java and C++ is around 1MB, which is relatively small compared to Heap memory. `StackOverflowError` occurs when an app hits its stack memory limit.
- **Heap**: Android uses this for dynamic memory allocation. The heap is a piece of memory where the system allocates Java/Kotlin objects in no particular

order. Android runtime limits the heap size for each running application to ensure a smooth multitasking experience. There is a variation of the heap size limit among devices and it depends on how much RAM a device has. If your app hits this heap limit and tries to allocate more memory, `OutOfMemoryError` will occur, and will terminate your app.

This is what the memory allocation looks like for a simple application:



- **Garbage Collection**: Garbage Collection is an action taken by the system to avoid memory-related issues such as `StackOverflowError` or `OutOfMemoryError`. When the system determines that any program or app no longer uses a piece of memory, it frees the memory back to the heap, without user intervention. The system conducts:
  - Search for data objects that are no longer referenced and you can't access them.
  - Reclamation of the memory occupied by those objects.

  This process cleans up and reclaims unused memory.

Why You Should Profile Memory

Garbage collection is a necessary process for memory management, but if it happens too often, it can negatively impact your app's performance. The system has to pause the app's code to let the Garbage Collector do its job. Normally, this process can be quite fast and imperceivable from the user's perspective.

If you've coded your app in a way that's not very memory efficient and allocates memory faster than the system can collect it, you'll notice your app is sluggish and skips frames.

In such cases, your code flow may force garbage collection events more often or make them last longer than usual. That slows down the rest of the system.

Eventually, the system might kill your app process to reclaim the memory and maintain a functional multitasking environment.

You should profile your app memory to avoid these problems, and that's where the **Memory Profiler** comes in handy!

Overview of Memory Profiler

Android Profiler is a combination of various tools that provide real-time information about your app, such as memory allocation and resource usage.

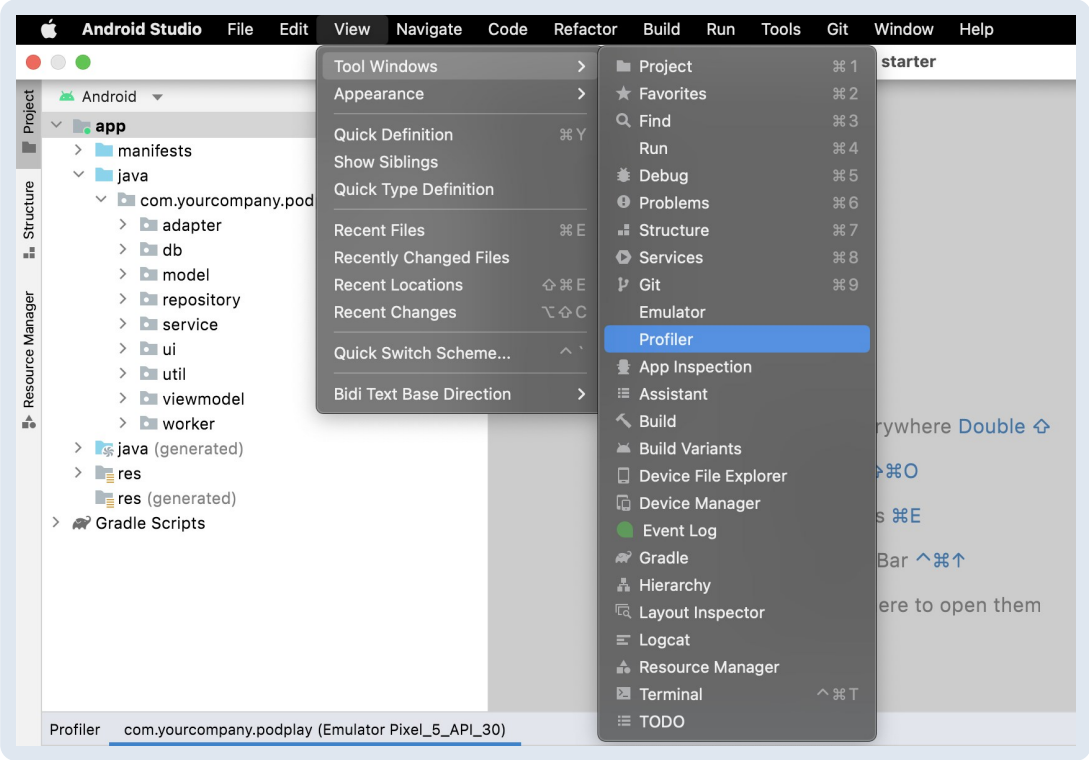
The Memory Profiler is a component of the Android Profiler. With Memory Profiler, you can:

- View the real-time status of allocated objects and garbage collection events on a timeline.
- Initiate garbage collection events.
- Capture heap dumps.
- Record memory allocations.
- View the stack trace for each allocation.

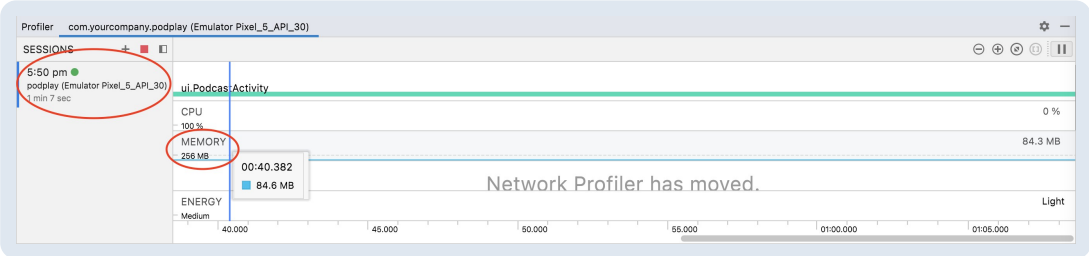
And more...

Running Memory Profiler

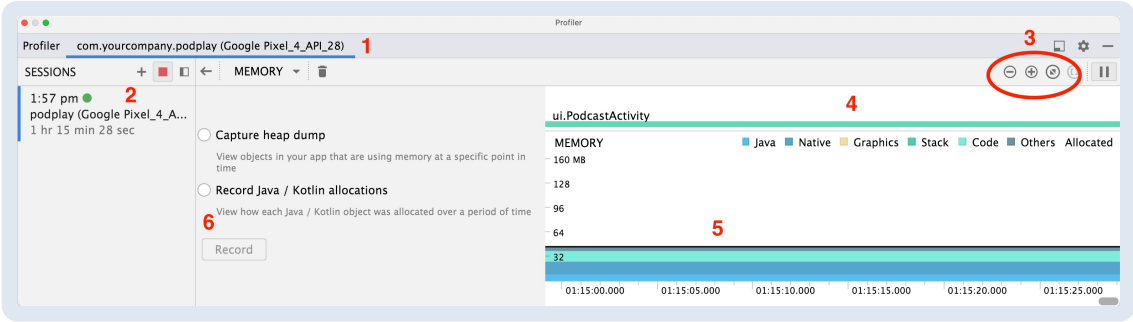
1. Open the **Podplay** starter project and run the app on an emulator or connected device using API level 26 or higher.
2. Select **View ▸ Tool Windows ▸ Profiler** from the menu bar.



3. This will start a new profiling session for PodPlay from the current time. Select **MEMORY** from the right pane as highlighted below:



This will open the actual Memory Profiler toolset from Android Profiler. Take a minute and have an overview of each of the highlighted sections as numbered below:



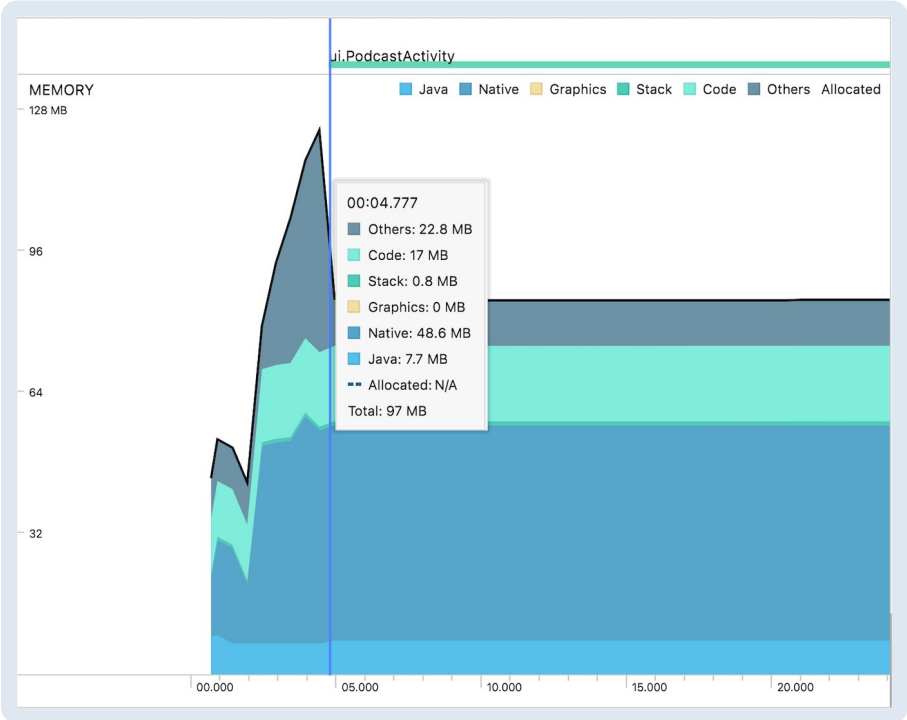
1. The **process and device** that you are currently profiling using Memory Profiler. You can see your app's package name here.
2. The **Sessions** pane shows your current session, if any. This pane can save Profiler data as sessions until you quit Android Studio. The sessions pane allows you to:

- Start a new profiling session.
- Stop adding data to the current session.
- Import a trace exported from any previous session.

3. The **zoom-in/out buttons** control how much of the memory timeline to view or jump to the real-time updates.
4. The **event timeline** shows user inputs or actions such as volume changes or screen rotations performed while profiling the app.
5. The **memory graph** displays memory that each category uses in different colors, i.e., Java, Native, Graphics, etc. The horizontal axis represents the passage of elapsed time since you started profiling. You can see the number of allocated objects using the numbers on the vertical axis.
6. Options to **record memory allocations** or **capture heap dumps**. You'll know how to use these features soon. The options in this section can vary depending on your device's or emulator's API version.

Memory Count

To see overall memory usage by PodPlay at any point since the app launched, put the **cursor** above the event timeline. You'll see the memory count of your app segmented into several categories as follows:



- **Java**: Memory from objects that Java/Kotlin code has allocated.
- **Native**: Allocated memory from C/C++ code objects.
- **Graphics**: Memory to display pixels to the screen. This is a memory shared

with the CPU, not dedicated GPU memory.

- **Stack**: Memory used by both native and Java stacks in your app. When your app invokes a method, it creates a block in the stack memory to hold local primitive values and references to other objects in this method. This normally relates to how many threads your app is running.
- **Code**: Memory used for code and resources such as dex bytecode, **.so** libraries and fonts.
- **Others**: Memory that the system doesn't know how to categorize.
- **Allocated**: The number of Java/Kotlin objects your app has allocated. Objects allocated by C/C++ code aren't counted here.

> **Note**: Even if you're not using C/C++ in your app, you might see some native memory used because the Android framework uses native memory to handle various tasks, such as loading image assets, though the SDK methods for those were in Java or Kotlin.

The numbers you see are based on all the private memory chunks that your app has consumed. This count doesn't include memories shared with the system or other apps.

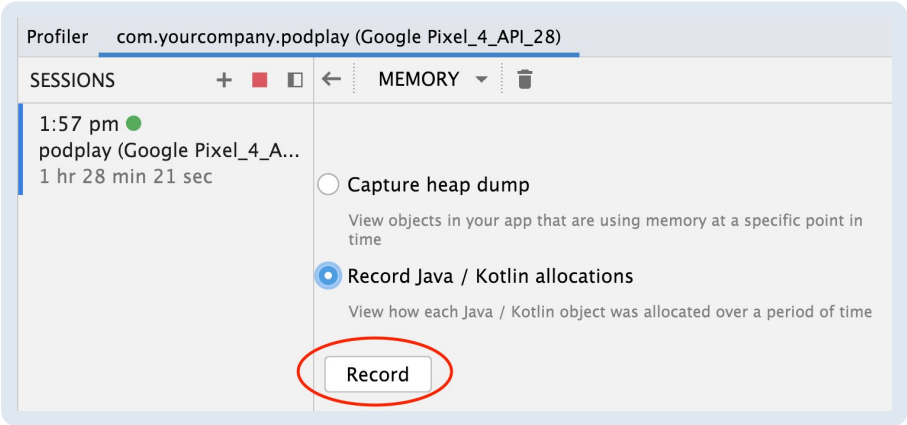
Tracking Memory Allocations

The memory allocations panel helps you see each object and JNI reference in your memory:

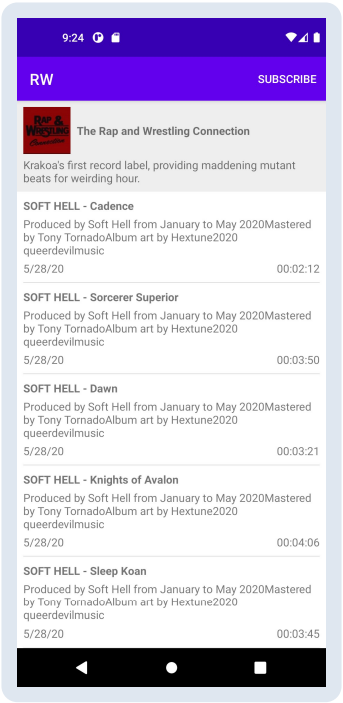
- The types of objects allocated and how much space they're using.
- The stack trace of each object allocation, including the information about their corresponding threads.
- When the objects were deallocated, this is only available for devices with Android 8.0 or higher.

Recording Memory Allocation

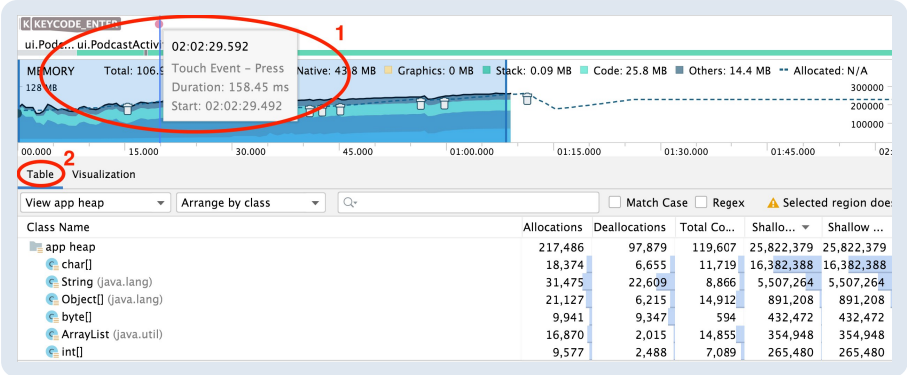
You'll now see how to analyze when you use PodPlay and objects the app creates in memory. Open the **Memory Profiler**, select **Record Java / Kotlin allocations** and click **Record**:



Then enter something into the search bar in `PodcastActivity`. That'll show a list of podcast channels based on your search query. Tap any item from the list, it will display details about the selected channel in `PodcastDetailsFragment` as follows:



The Memory Profiler recorded all your actions and memory allocations. Your recorded session will look like this:



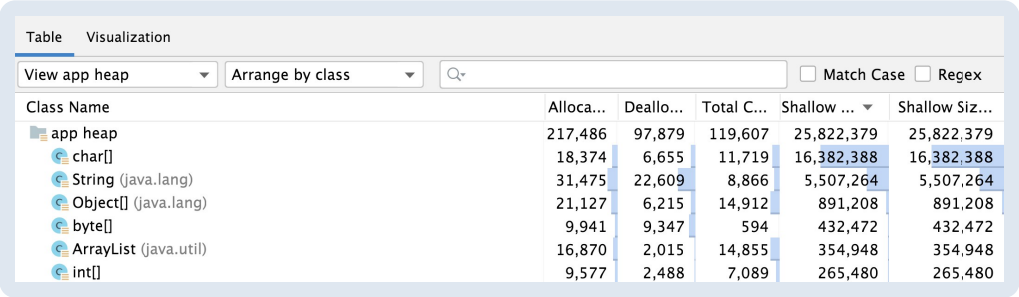
This might draw your attention to two focus areas:

1. The events you performed logged there, such as keyboard inputs or taps. That's how an **event timeline** displays user events in real time for a session.
2. A detailed table listing memory allocations underneath the event timeline.

Now it's time to learn more about the memory allocations table.

Memory Allocation Table

The table displays a list of allocated objects, grouped by class name and sorted by their heap count as it's shown below:



The **Class Name** column explains itself. Look at the other columns in this table:

- **Allocations**: Total number of objects currently allocated in memory for a specific class type in your session.
- **Deallocations**: Number of objects of that class that's already deallocated or garbage-collected until now.
- **Total Count**: Number of total objects remaining for that class in the session.

> Total Count = Allocations - Deallocations

- **Shallow Size**: The total size in bytes of all instances of the class in the memory.

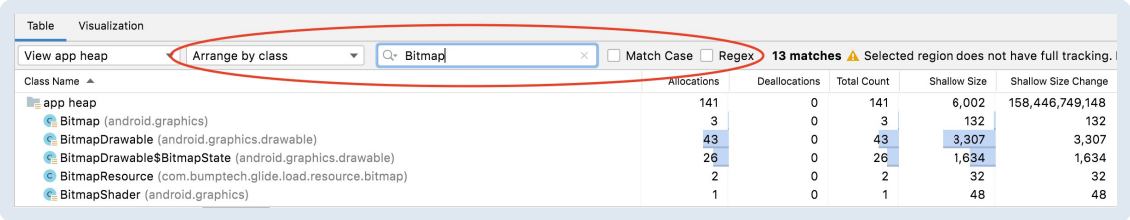
> Shallow Size = Memory consumed by one object * Number of objects.

- **Shallow Size Change**: Difference in Shallow Size since the last garbage collection or memory deallocation happened.

It's easy to browse the list to find objects with unusually large heap counts, for example, Bitmaps. Click the **Shallow Size** column header to sort the list by largest memory allocating classes.

> **Note**: Click any column header in the list to sort results by that field.

To find known classes quickly, enter a class or package name in the search field:



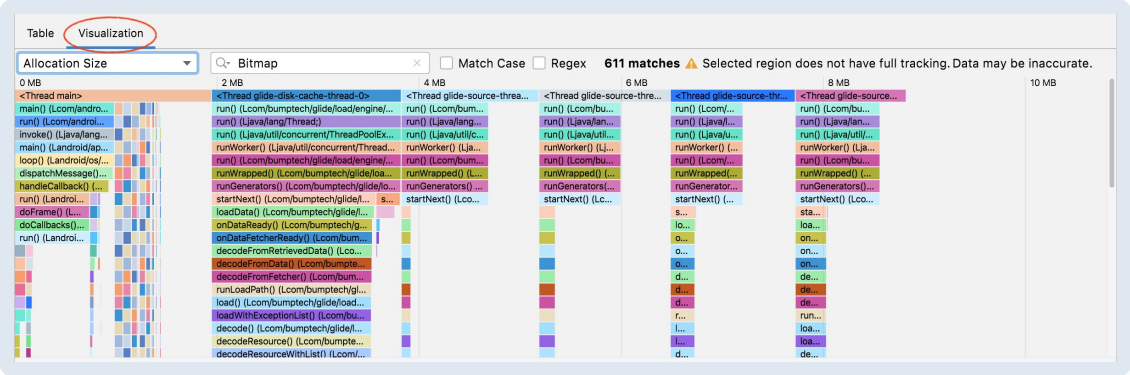
If your search query is case-sensitive, check the box next to **Match Case**

Check the box next to **Regex** if you want to use regular expressions.

You can also search by package or method name if you select **Arrange by package** or **Arrange by callstack** from the dropdown menu on the left of the search field.

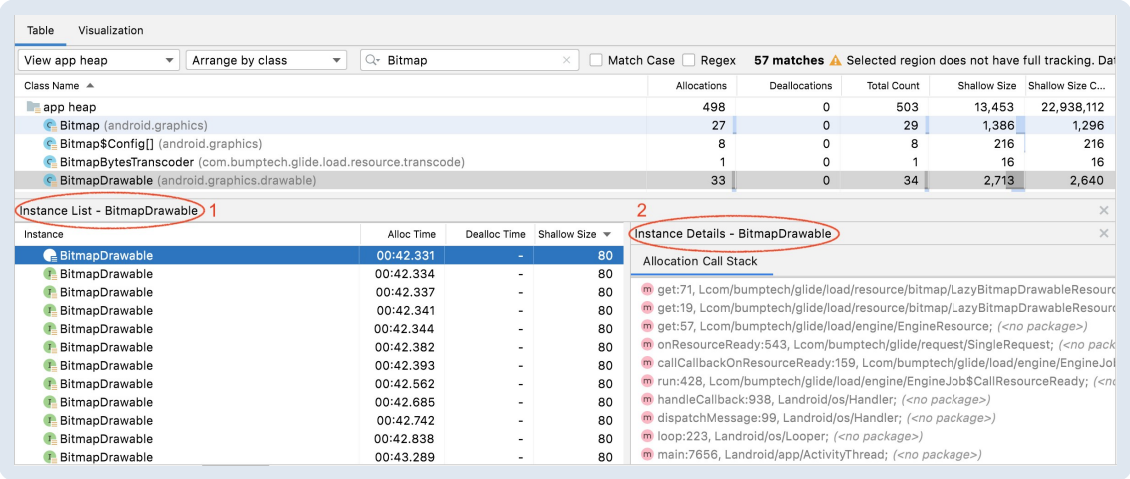
The PodPlay app uses the Glide image loading library to download images, also known as Bitmaps, from remote servers and handle the image loading on separate threads. That keeps PodPlay's Main thread free and always responsive to user interactions. You might be wondering what that looks like in memory.

Switch to the **Visualization** tab in the highlighted area below, and you'll be amazed to see the organization of all the Bitmap-related threads in the memory:



The above image shows the Main Thread and other threads from Glide to load Bitmaps with their method call-stack.

To see even more details from the list, switch back to the **Table** tab and click a class name, for example, **BitmapDrawable**. A new pane will open below the list as shown here:

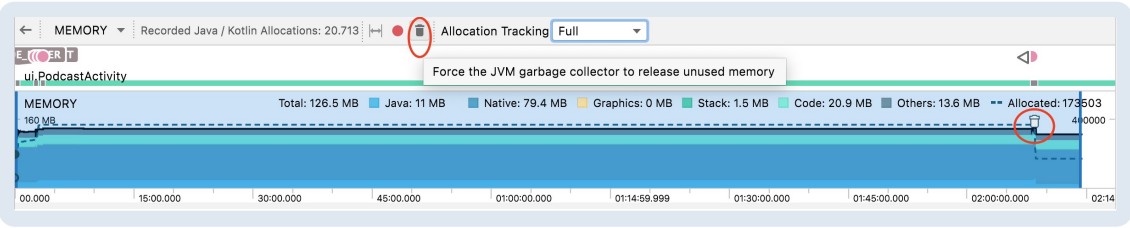


This pane is in two main segments:

1. **Instance List**: A list of all the instances of your selected class, in this case - BitmapDrawable, including their allocation/deallocation time and the memory each instance consumed, namely, **Shallow Size**.
2. **Instance Details**: This section shows the allocation of that instance and which thread it's in. This is the call stack for that object. From the above image, you can see the BitmapDrawable being fetched with a `get()` call from the Glide library, all from the **main** thread.

Forcing Garbage Collection

At this point, you might want to see how garbage collection impacts memory allocation. To force a garbage collection, click the **Trash** icon highlighted below:



That'll free up some memory. It will mark the moment garbage collection occurred in the timeline as shown above. See if you can find the difference in memory allocation using the skills you just gained!

Improving Profiling Performance

To improve performance during profiling, the Memory Profiler samples memory allocations periodically. You can change this by using the **Allocation Tracking** dropdown beside the **Trash** icon shown above.

The options available are as follows:

- **Full**: This captures all object allocations in memory. A downside of this option is that if you have an app that allocates many objects, you may observe visible slowdowns with your app while profiling.
- **Sampled**: Samples object allocations in memory at regular intervals. This option is selected by default as it has less impact on app performance while profiling. Apps that allocate many objects over a short period may still suffer from slowdowns.
- **Off**: Stops tracking your app's memory allocation.

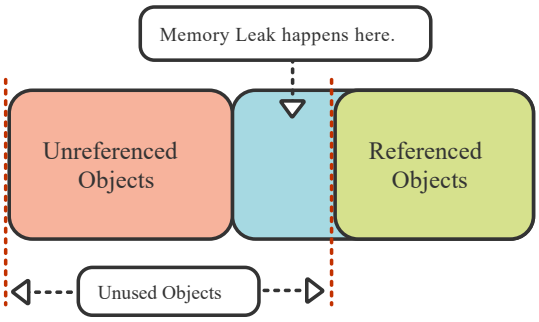
You can end the sessions now by clicking **Stop** in Memory Profiler.

Memory Churn And Memory Leaks

You just learned to force garbage collection by yourself while using the Memory Profiler. Android Runtime performs garbage collection periodically in a typical scenario, but what if there's a case that "forces" the system to do it?

Forced garbage collection occurs when the app allocates but also has to deallocate objects in a short period. It can, for example, happen if you allocate heavy objects like Bitmaps in loops. In each iteration, they'll keep saturating your heaps. The system not only has to allocate a large object but it also has to deallocate it from the previous iteration, so it doesn't run out of memory, resulting in more garbage collections. This situation is a **Memory Churn**. Users may notice stuttering or slowdown in the app because of this frequent garbage collection.

A **Memory Leak** happens when your code allocates memory for objects but never frees that memory or is unable to deallocate it. Over time, the memory allocated for these objects turns into a large, immovable block, forcing the rest of the app to operate in what's left of the total heap memory. If this continues, eventually, the app can run out of memory and crash.



Memory leaks can be huge and obvious, such as when your app is trying to load a high-resolution Bitmap that's larger than the available memory. Some

memory leaks can be hard to find, so the user will only notice the app lagging over time. Capturing heap dumps and analyzing them can help to figure out such memory leaks.

Identifying and Improving Memory Performance: Detecting Memory Leaks

You can analyze the current state of the memory by performing a heap dump. It shows which objects in your app are using memory when you capture them. After a user session, a heap dump can help identify memory leaks by showing objects still in memory that should no longer be there.

A captured heap dump shows you the following:

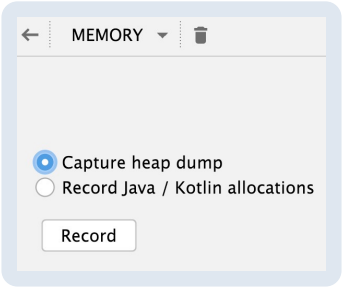
- The types of objects your app has allocated, and how many of each type.
- Memory used by each object.
- Where references to each object are being held in your code.
- The call stack for where an object was allocated.

> **Note**: You can capture heap dump while recording allocations and get a referent call stack with Android 7.1 or higher only.

Capturing a Heap Dump

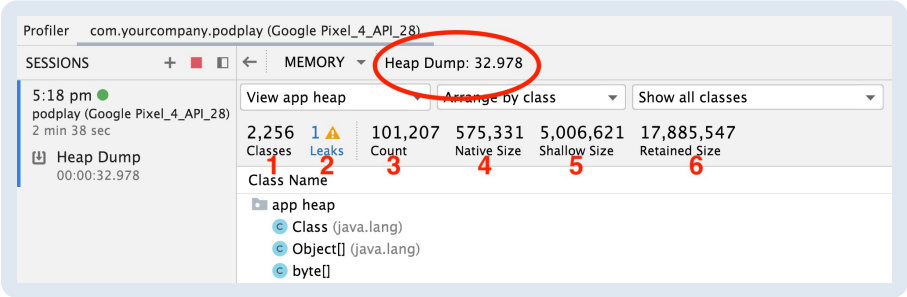
To capture a heap dump, perform these actions as you've previously completed:

1. Run the app on a device.
2. Run Android Profiler and, in the **type** dropdown, switch to the Memory section.
3. Enter something into the search bar.
4. Select any item from the podcast channel's list and go to `PodcastDetailsFragment` to see details.
5. Tap the **Back** icon and go back to the list screen.
6. In the Memory Profiler, select **Capture heap dump**, then click **Record**.



While dumping the heap, you may observe the amount of memory getting increased temporarily. You should expect this because the heap dump occurs in the same process as your app and requires some memory to capture the data.

Below the timeline, you'll see the heap dump as follows:



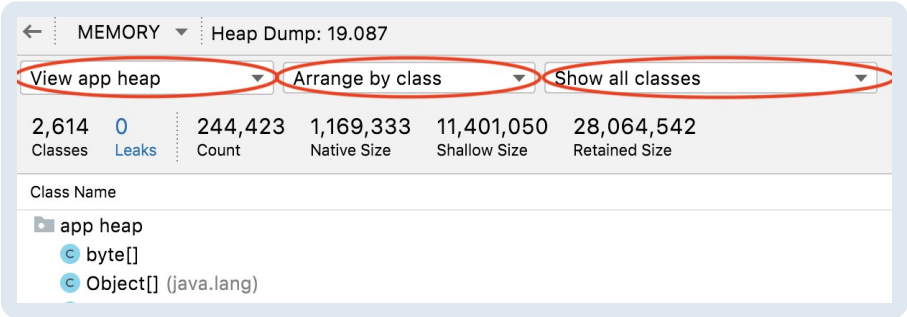
Take a moment to understand what each of the components in this panel offers.

The red circle on top displays a timestamp. It shows the elapsed time from the app launch to the moment you captured the heap dump.

Now, look at the numbered areas. You can see the explanation for each of them below:

1. **Classes**: Number of different class types in the heap.
2. **Leaks**: Number of potential memory leaks; you'll jump into this soon!
3. **Count**: Total number of allocations in the heap.
4. **Native Size**: Total amount (in bytes) of native memory used by the object type. You'll see memory allocation for some Java objects here because Android uses native memory for some framework classes, such as `Bitmap`.
5. **Shallow Size**: Total amount (in bytes) of Java memory used by this object type.
6. **Retained Size**: Total size (in bytes) of memory being retained by all instances of this class.

Next, look at the dropdown menus.



The first dropdown menu on the left lets you choose which heap to inspect:

- **View all heaps**: Shows data for all the heaps when the system specifies no heap.
- **View app heap**: The primary heap where your app allocates memory.
- **View image heap**: The heap for the system boot image. This contains classes that the app preloads during boot time. Allocations here never change or go away.
- **View zygote heap**: The copy-on-write heap where an app process is forked from in the Android system.

With the dropdown menu in the middle, you can choose how to arrange the allocations:

- **Arrange by class**: Groups all allocations based on the class name. This the default selection.
- **Arrange by package**: Groups all allocations based on the package name.
- **Arrange by call stack**: Groups allocations into their corresponding call stack. This option only works if you've captured the heap dump while recording allocations.

The next dropdown menu lets you filter displayed classes based on the below criteria:

- **Show all classes**: Displays all the classes, including base classes, regardless of whether it's initiated from the system or the app.
- **Show activity/fragment Leaks**: Displays class names of the `Activity` or `Fragment` if it's creating a memory leak.
- **Show project classes**: Displays classes only from the source code in your project.

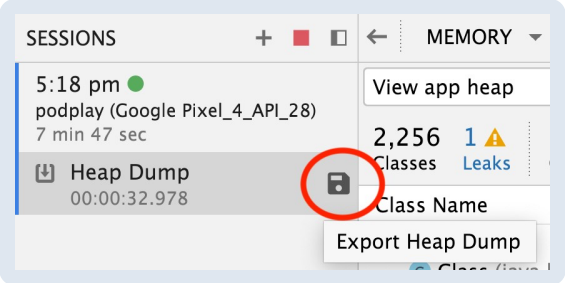
Saving and Importing a Heap Dump

After you capture a heap dump, the data is visible in the Memory Profiler as long

as the profiler is running. You lose the heap dump as soon as you exit the profiling session.

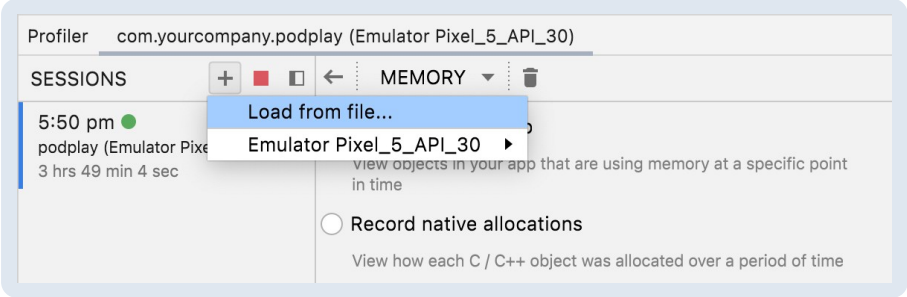
So, before fixing the memory leak, save this heap dump so that you can compare after applying the fix.

To save a heap dump, hover the cursor on the **Heap Dump** entry in the **Sessions** pane, and click the **save** icon as highlighted below:



An **Export As** dialog will appear. Save the file in your desired location. The saved file will have `.hprof` extension.

To import the saved file, click **Start new profiler session ▸ Load from file...**



Then, choose the file from the browser window.

> **Note**: You can also import a `.hprof` file by dragging it from the file browser into the Memory Profiler panel.

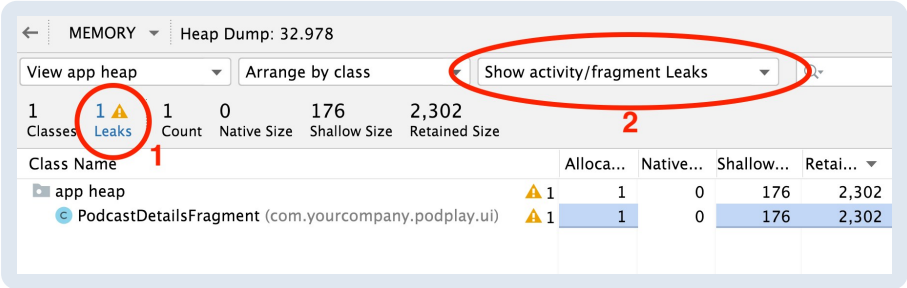
Fixing a Memory Leak

You might be yearning to fix the memory leak you've seen during heap dump. Don't worry, you can easily filter on profiling data that the Memory Profiler thinks might induce memory leaks in your app.

The Memory Profiler allows you to filter:

- `Activity` instances that have been destroyed but are still being referenced from somewhere.

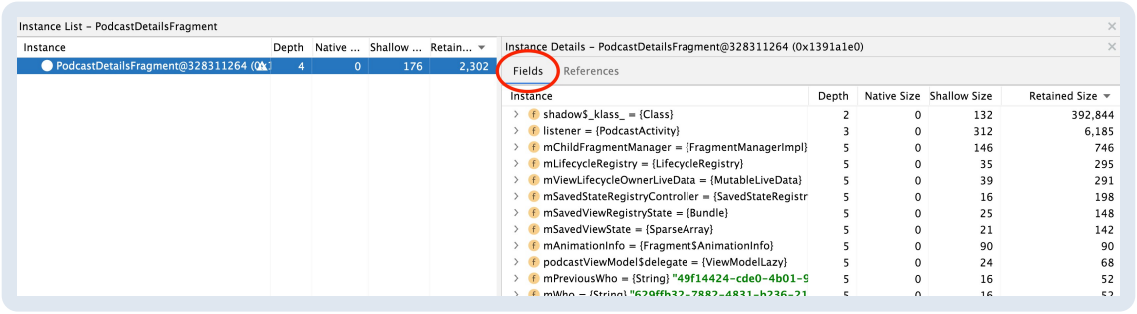
- `Fragment` instances that have been destroyed or don't have a valid `FragmentManager` but are still being referenced.



There are two ways you can see memory leaking `Activity` or `Fragment` right away:

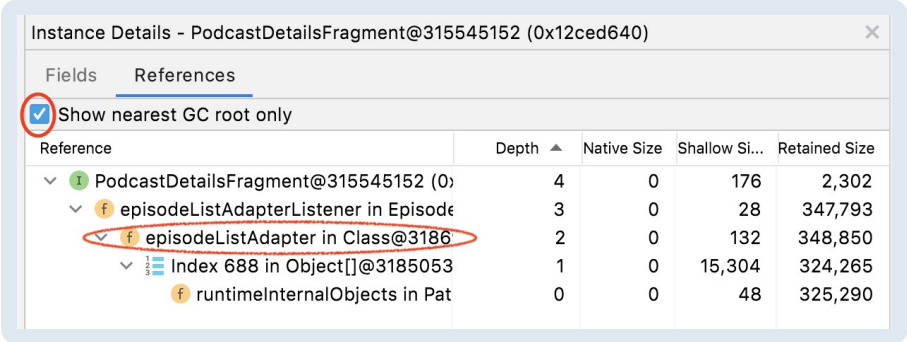
1. By clicking on the number of memory leaks.
2. By selecting **Show activity/fragment Leaks** from the dropdown menu.

Click `PodcastDetailsFragment` from the list. That'll reveal more detail below about the memory leaking instance as follows:



Looking at the **Instance Details** section on the right pane indicates that the whole `PodcastDetailsFragment` instance might be retaining in the memory even after it's supposed to be destroyed, causing the memory leak!

Now switch to the **References** section, then check **Show nearest GC root only**:



That gives you enough clues that there's something wrong with the `episodeListAdapter` instance.

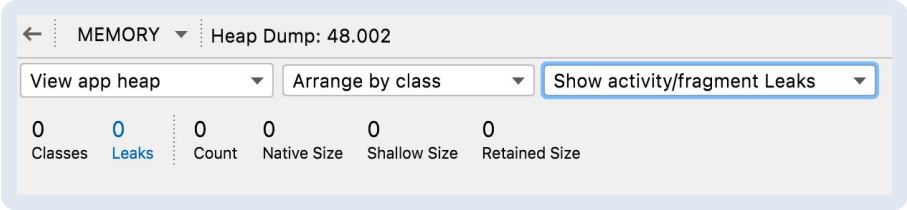
Note: To know more about GC Root (Garbage Collection Root), you can refer to Android Memory Profiler: Getting Started.

To figure out what's wrong, open **PodcastDetailsFragment.kt** in the IDE and look at **line 59** where `episodeListAdapter` is declared.

Do you know what's wrong there? `episodeListAdapter` was declared within the `companion object` block, possibly a silly programming error. This creates a static `episodeListAdapter` instance. Even though when you go back to the previous screen, and `PodcastDetailsFragment` is supposed to be destroyed, the static instance is still being kept in the memory and referencing the class.

Well, now you know the root cause of the memory leak. To fix that, move the `episodeListAdapter` out of `companion object` and place it on the top of the class.

Now re-launch the app, and capture a heap dump with the same steps mentioned in that section. You'll be glad to see that the memory leak's finally gone!



Memory Profiling: Tips & Tricks

Memory Leak is very common since developers can't see it happening while developing the app. That's why memory profiling is crucial.

You should stress your app code and try forcing memory leaks, especially before you release the app. Below are some ways you can provoke memory leaks in your app:

- To run the app for an extended period before doing a heap dump. The smaller the leak, the longer you'll need to run the app to find it.
- Rotate the device multiple times in different activity states. The system recreates the `Activity` / `Fragment` during a screen orientation change. If your app has a reference to one of those objects, the system can't garbage collect it, and it'll mark it as a memory leak.
- Switch between your app and another app while using different UI conditions. For example, navigate to the Home screen, then switch back to your app several times.

As a developer, to avoid creating memory leaks and churns, you should be mindful to *not*:

- Leak **View** objects in any situation, they're memory-heavy!
- Reference views from outside of the UI thread.
- Reference a view in an async callback, you can't free that view until the task is done.
- Reference views from static objects. Static objects stick around for the lifetime of the application.
- Put views into collections, that's very expensive in terms of memory allocation.
- Allocate objects in inner loops. Do it outside the loop, or redesign to skip allocation in the first place.
- Allocate objects in `onDraw()` of any custom views. `onDraw()` is called on every frame, taking a heavy toll on the heap memory.

Key Points

- Memory Profiling keeps app performing at its max. Profiling your app at different stages of development can result in finding memory leaks early.
- Memory Profiler offers a complete toolset to view memory allocations, method stack-trace and memory leaks if any.
- Capturing and analyzing heap dumps frequently helps to understand memory allocation and improvement points and find memory leaks.

Where to Go From Here?

Though memory management is a complex and vast topic, you learned a lot about memory allocations, garbage collection, and analyzing memory allocation from a heap dump of fixing memory leaks! To learn more on this topic, check out the tutorials and videos below:

- Memory Leaks in Android
- Google Developer Training: Memory in Android
- Manage Your App's Memory
- Android Performance Patterns

In the next chapter, you'll learn to profile network activity, so good luck on your next adventure.

11 Profile Network Activity

Written by Zahidur Rahman Faisal

You've previously learned how to look into memory footprint using the Android Studio Memory Profiler as a part of your debugging process. In this chapter, you'll continue your quest and learn to inspect network traffic using the **Network Inspector** bundled with Android Studio.

Your smartphone performs plenty of network operations every day. From uploading a selfie to your social media account to chatting with someone, all of these tasks are some sort of network operation. Smartphone apps like PodPlay let you subscribe to your favorite podcasts and download the latest updates from your subscribed channel, making network calls, all for your convenience.

This is cool but can be overwhelming for your device if the app performs frequent, unnecessary network operations. To minimize your network data usage and resource consumption, you need to inspect your network activity to keep it optimal.

In this chapter, you'll learn how to:

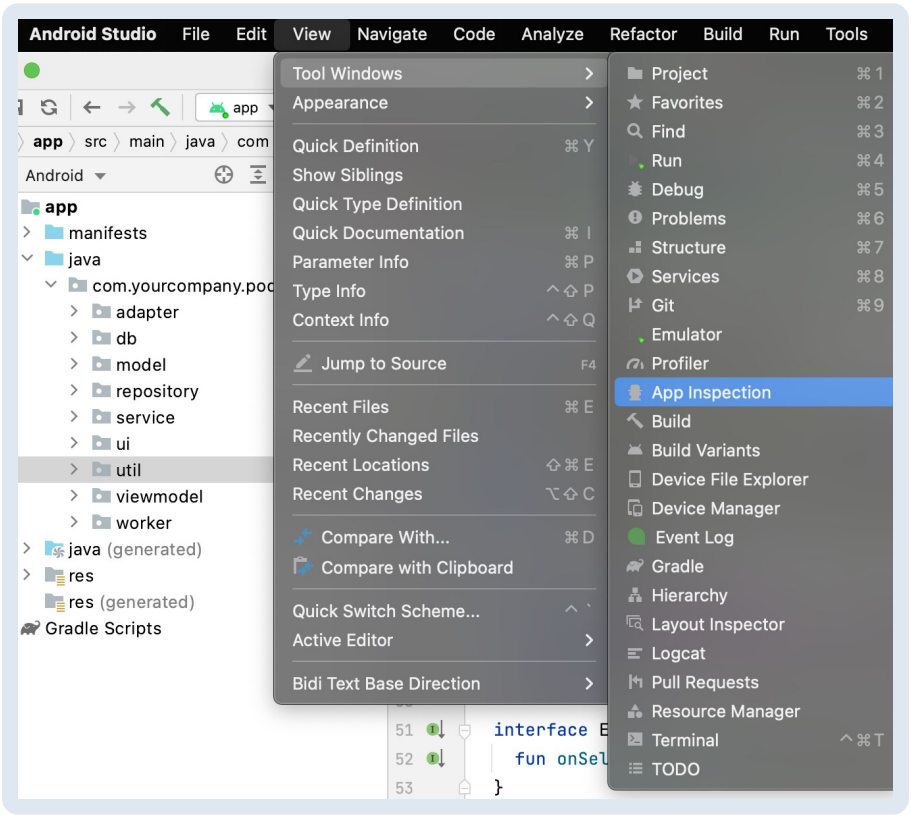
- List the network operations your app performs over a timeline.
- Find states of a network operation from the aspect of connection status or threading.
- View details of network requests and responses.

The Network Inspector

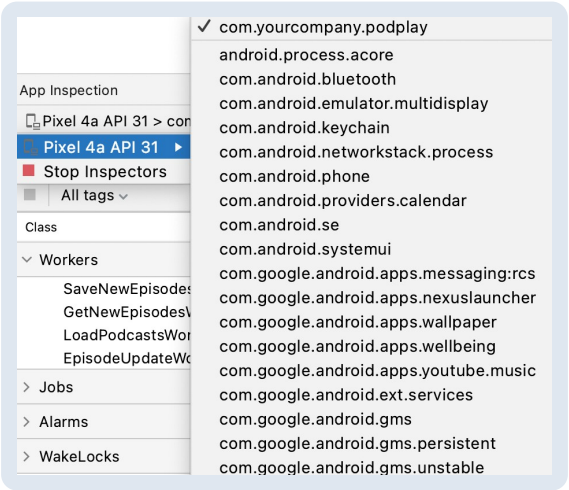
The **Network Inspector** is your one-stop solution to spy on network operations executed by PodPlay. It lets you examine how and when the app transferred data with a timeline in real time.

Follow these steps to bring up the Network Inspector:

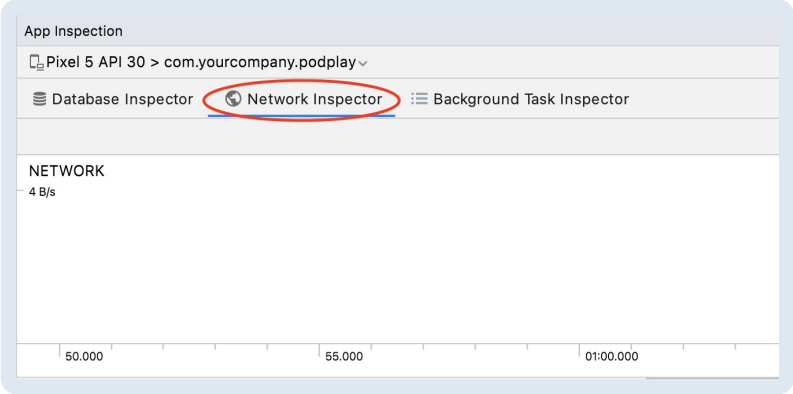
1. Open the **Podplay** starter project and run the app on an emulator or connected device using API level 26 or higher.
2. Select **View ▸ Tool Windows ▸ App Inspection** from the menu bar.



3. In the **App Inspection toolbar**, choose your device. Then, select **com.yourcompany.podplay** from the running app process in the dropdown menu.



4. Select **Network Inspector** from the tabs.

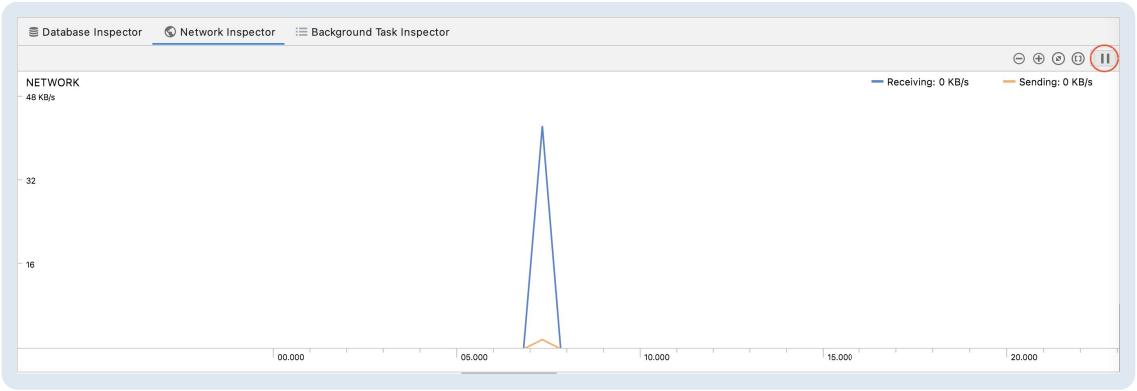


You won't see much at this point apart from a blank timeline moving at the bottom of the Network Inspector panel. That's your network timeline, and soon you'll be able to make the most out of it.

Using the Network Timeline

The network timeline is a real-time representation of network activities from your app. It starts from the moment you launch the app and moves as time passes. To see it in action, search for a podcast channel following these steps:

1. Tap the search icon.
2. Put "RW" for the search input and press **Return**.



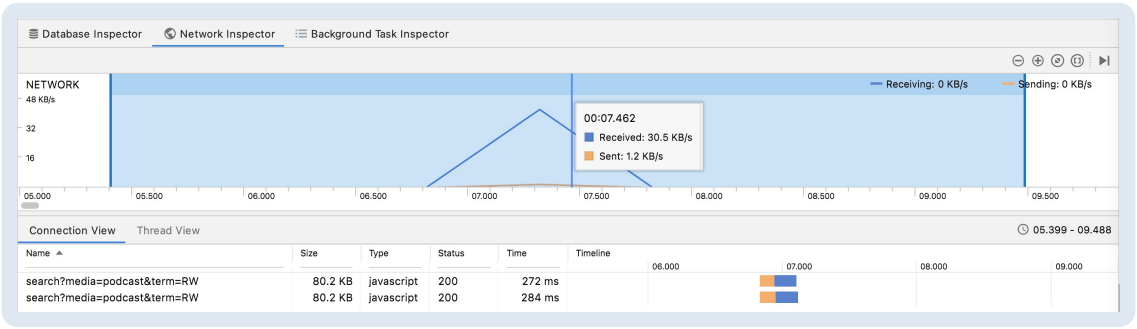
You'll see a spike in the network timeline as above. Click the **Pause** button in the top-right corner of the Network Inspector panel. This will stop your network request (this spike) from moving away with time.

Now, take a moment to analyze what's going on with the network request to get a deeper understanding.

The Network Inspector displays a few more buttons that give you control over the timeline. The utility of these buttons is as follows:

1. **Zoom Out**: Zooms out the timeline, and displays a longer period of time to allocate more requests, if there are any, in the timeline view.
2. **Zoom In**: Zooms in the timeline displaying a shorter period of time in the timeline view.
3. **Reset Zoom**: Resets any zoom-out or zoom-in over the timeline view and reverts to the default zoom state.
4. **Zoom to Selection**: This allows you to zoom into any selected interval from the timeline. This is helpful to focus on a specific time frame you select to identify a network request which occurred within that period.

As you can see, your network request has been made in a period between 5 to 10 seconds after launching the app, select that interval from the timeline and click **Zoom to Selection**. This will reveal another panel below the timeline, with two tabs — **Connection View** and **Thread View**.

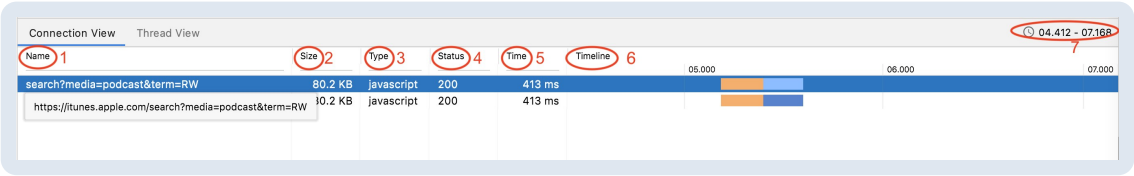


These tabs expose detailed information about the network requests you've made within the selected interval.

> **Note**: Put your cursor over any point on the network request to view sent and received data at that moment in the timeline.

Connection View

The **Connection View** tab displays files or network requests sent or received during the selected period in the network timeline, irrespective of threads. You can inspect each request's name or query, size, type, status and transmission duration.



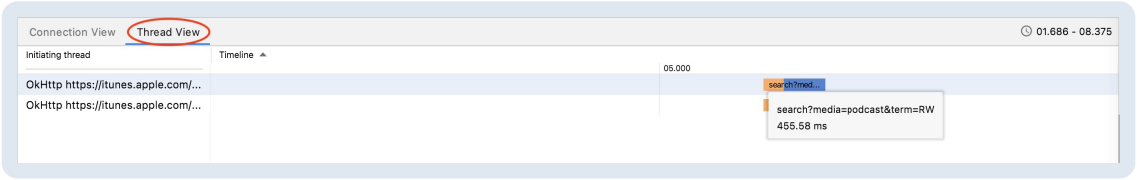
Here is the summary:

1. Name of the network request. This column shows the request URL except for the hostname part. Hover the cursor over the request name to see the full request URL.
2. Size of total data transferred for this network operation.
3. The type of content received from the request. In this case, it's Javascript in JSON format.
4. **HTTP Status Code** for this request. It indicates whether a specific HTTP request has been completed successfully or had any errors. The selected request was successful and returned with a proper response; hence it's showing *200*. You can learn more about HTTP Status Code in different scenarios here.
5. The time taken for the whole network operation to complete. According to this, it took a total of *413 milliseconds* to execute the request and receive a response from the server.
6. Duration for the selected request in the timeline. You can see the network operation occurred for less than a second, 413 milliseconds to be specific, sometime between the 5th and 6th second of launching your app.
7. The range in the top-right corner denotes the selection time frame from the network timeline. In this case, you displayed network operations executed from the 4th to the 7th second in the timeline.

> **Note**: You can sort the list in **Connection View** by selecting any of the column headers. For example, to sort the network requests based on the size of data transferred, select the **Size** column.

Thread View

The **Thread View** lists all CPU threads that have initiated a network activity from your app. Switch the tab from **Connection View** to **Thread View**, and you'll see a panel like this one below:



In the Thread View, you can see the OkHttp framework has been used to create and manage network requests from PodPlay. **OkHttp** handled the threading mechanism and concurrency for all network operations from within the app. You'll get more details on threading later in this chapter.

The Thread View contains the timeline column as well. With this cool feature, it's easy to visualize what happened on the thread for network operations between the 5th and 6th seconds.

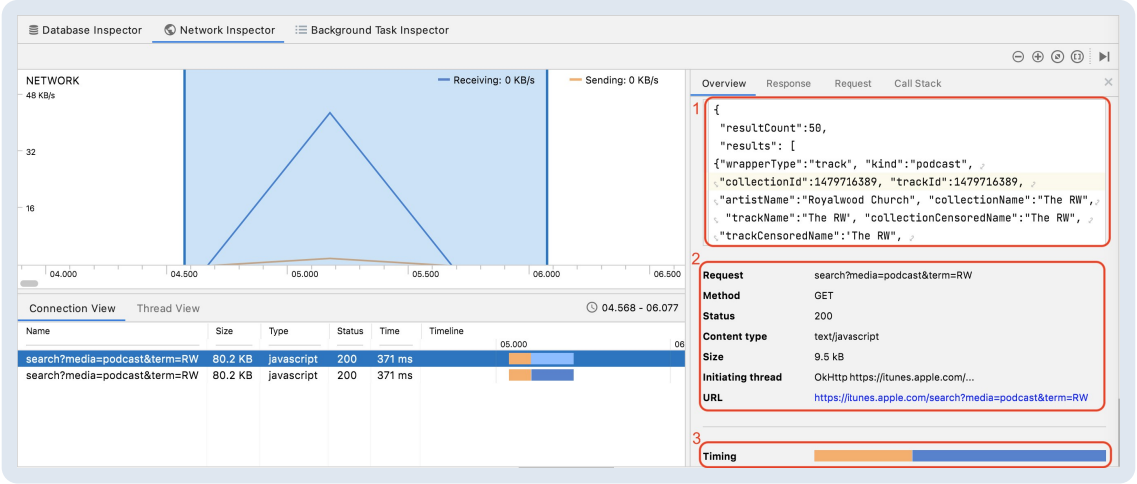
Again, hovering your cursor over the selected network thread will display the request parameters and duration for that operation, as shown above.

Inspecting Details

The Network Inspector offers even more details about the **Request**, **Response** and **Threads** for each individual network operation.

Overview

Switch back to the **Connection View** tab, then select the first row. You'll see a new panel on the right side:



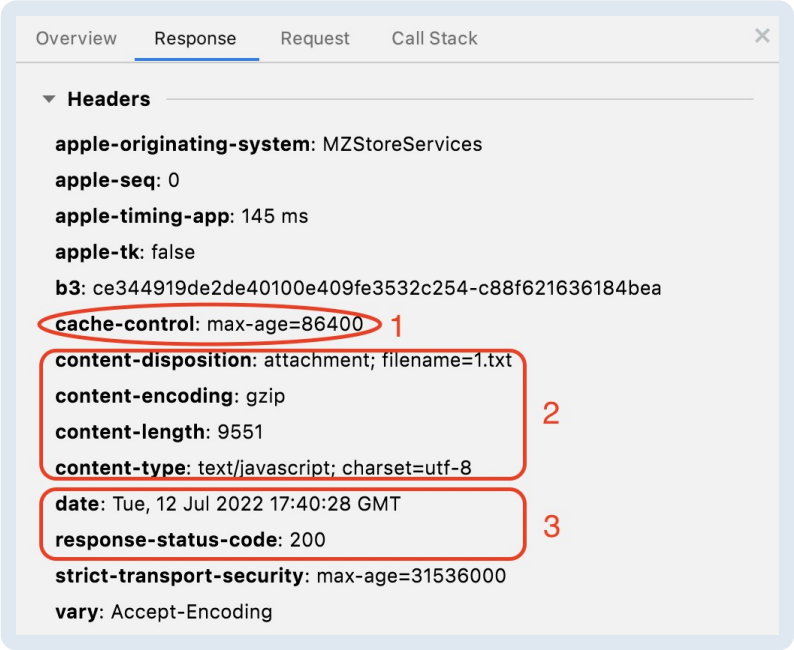
This is the **Overview** which holds a lot of interesting details about your network request. It contains 3 main sections:

1. Displays the response received from the server for this particular network operation. In this case, it's a JSON object holding an array of podcasts that include your search query "RW".

2. Presents details about the request made from the app. Looking at this section, you can extract info such as the full `URL`, the request method, whether it's a GET, POST, PUT or another type of request, the status code, content type or size of data exchanged for the request.
3. Shows a bar with an overview of exchanged data for this network operation. Don't let the name confuse you, this bar explains that the network operation initially sent a small amount of data to make the request, then received a larger portion of data over the time of execution. The *orange* segment is sent data and the *blue* segment is received data proportionally.

Response

Now, switch to the **Response** tab. The Response tab presents the response headers and the full response body. The response headers hold additional information about the server or how the response is cached:

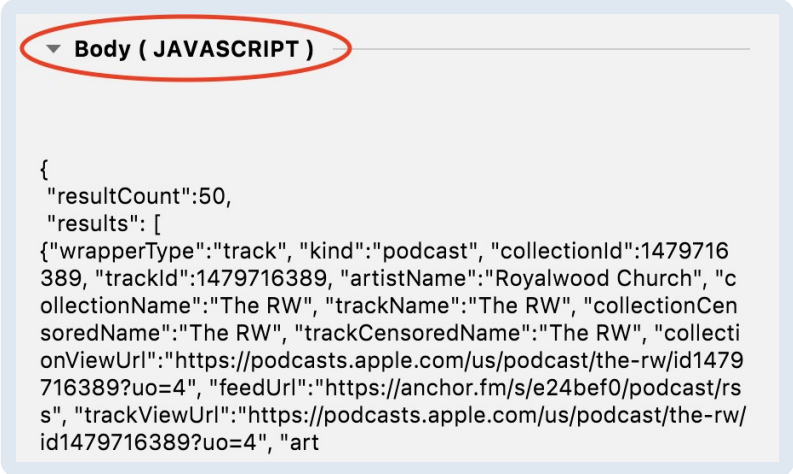


That's a lot of information! You might want to focus on a few important areas highlighted above:

1. **Cache**: The **max-age** in the **cache-control** header indicates that the response will be cached for 86400 seconds. Caching the response reduces server load and bandwidth transmitted, as well as improving loading times.
2. **Content**: Content headers contain the size, type or encoding info of the response object. You can see **content-length** is 9551 bytes. This is especially useful when you're downloading a file. The **content-type** is the **MIME type** of the content. It indicates **text** is the type and **javascript** is the subtype of the response.

3. **Status**: In most of the cases, you'll just need to check the **date** or **response-status-code** to verify the time you've received the response and if you've received it successfully. Status code 200 means it was successful.

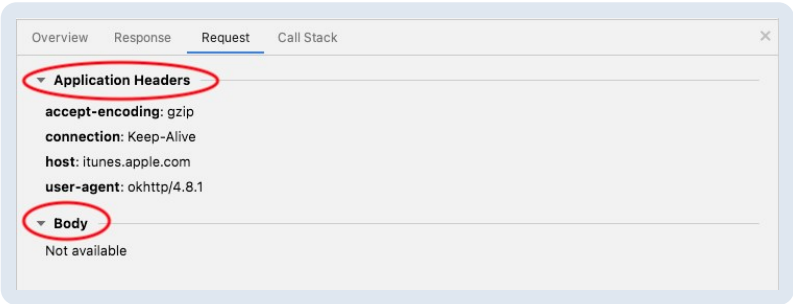
Scroll down a bit to find another detail:



Body is the most important section here, which displays the complete response. It receives the response as a JSON object in this case. The panel displaying it allows you to scroll down and check the full object.

Request

In the **Request** tab, you don't get as many details as in the **Response** tab. This tab has two sections: **Application Headers** and **Body**.



These 2 sections reveal the following information:

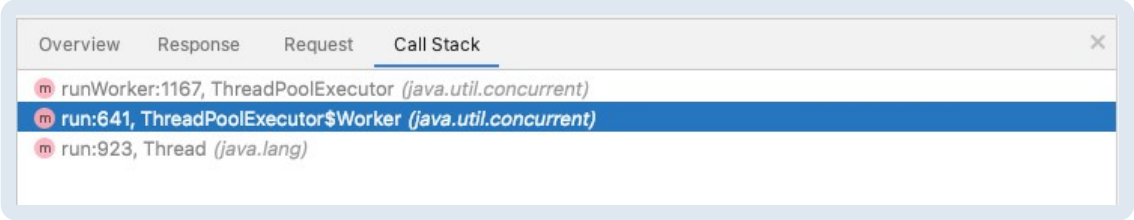
- **Encoding**: **Accept-Encoding** tells the server that the app can accept compressed output like `gzip` for the response.
- **Connection**: A default HTTP connection is usually closed after completing each request. **Keep-Alive** header, also known as *HTTP persistent connection*, is kind of a communication message between the app and server that says: "You may grab many files as long as the connection is alive." This configuration is handy if you request something that includes multiple files, such as a web page. Setting Keep-Alive transfers all the necessary files

through a single connection request instead of creating and closing multiple connections.

- **Host**: The domain name or the server's hostname you've requested. In this case, it's **itunes.apple.com**.
- **Agent**: **user-agent** represents a software or framework that retrieves, renders and facilitates interaction with the web content on behalf of the user, the PodPlay app. You can see PodPlay relies on the **OkHttp** library to create and manage network requests for you. It also displays that the version code of **OkHttp** library is 4.8.1.
- **Body**: Since it was a GET request, the request body isn't available in this case. You could see the request body submitted to the server in this section if it was a POST or PUT request. The pane is useful to verify if the request body submitted is what the server expects when creating a POST or PUT request.

Call Stack

The fourth and final tab in the network details panel is **Call Stack**. Switch to that tab, and you'll see a stack of threads that corresponds to the network operation as follows:

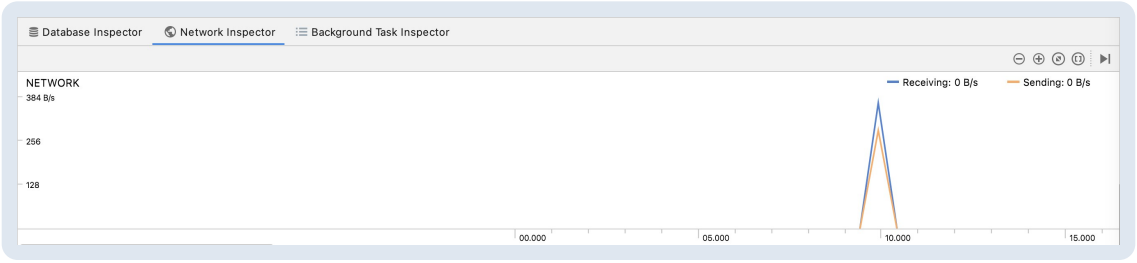


As you already know the **OkHttp** library is handling threading for the network operations. You can see it's running a Worker Thread using ThreadPoolExecutor. This ensures concurrency when there are multiple requests.

By inspecting the call stack, you can also tell that the threads for network operations, and worker thread, are running on top of the Java main thread, also known as the UI Thread. This isolates the network process from the main thread. That's the trick that keeps your app responsive while running an expensive network request in the background!

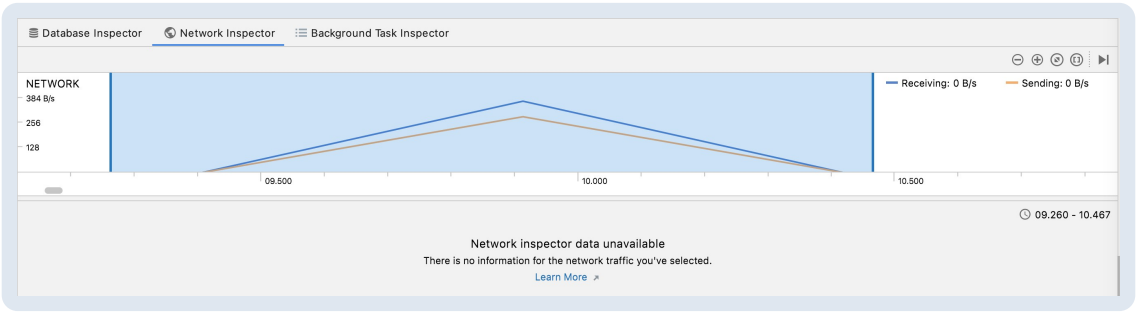
Tracing Unusual Network Traffic

Now you know how to inspect network operations using the tool-set offered by **Network Inspector**. Relaunch the PodPlay app, hold on and do nothing for 10 - 20 seconds, and you'll notice something odd in the network timeline:



What, a spike!? But you did nothing... right?

Apparently, there's some unwanted network operation made from PodPlay. Select the spiked segment from the timeline, and you'll be surprised to see that there's no information available:



That's strange, but this situation occurs when the **Network Inspector** detects traffic values yet can't identify any supported network requests. In such cases, you might want to log your network operations to see what's happening.

Logging Network Operations

Open **ItunesService.kt** and add these imports on top:

```
import com.yourcompany.podplay.BuildConfig
import okhttp3.OkHttpClient
import okhttp3.logging.HttpLoggingInterceptor
import java.util.concurrent.TimeUnit
```

Then replace the `companion object` as follows:

```
companion object {
 val instance: ItunesService by lazy {
   // 1
   val client = OkHttpClient().newBuilder()
       .connectTimeout(30, TimeUnit.SECONDS)
       .writeTimeout(30, TimeUnit.SECONDS)
       .readTimeout(30, TimeUnit.SECONDS)
   // 2
   if (BuildConfig.DEBUG) {
       val interceptor = HttpLoggingInterceptor()
       interceptor.level = HttpLoggingInterceptor.Level.BODY
       client.addInterceptor(interceptor)
   }
```

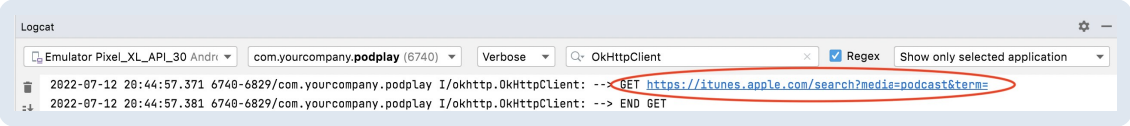
```
    // 3
    val retrofit = Retrofit.Builder()
        .client(client.build())
        .baseUrl("https://itunes.apple.com")
        .addConverterFactory(GsonConverterFactory.create())
        .build()
    retrofit.create(ItunesService::class.java)
  }
}
```

The above code has three changes to the previous implementation, it:

1. Creates an OkHttpClient to send HTTP requests and read responses.
2. Attaches a HttpLoggingInterceptor for logging HTTP request and response data when your app is in debug mode.
3. Uses the **OkHttpClient** object with your **Retrofit** builder, which is responsible for communicating with the server.

Relaunch PodPlay and look into the **Logcat** from the IDE. Search for the term "OkHttpClient".

Now you'll be able to see the request and response made from the app getting logged:



Highlight the request from the logs you have. You can clearly see the GET request ends with `term=`, which means the request has been made with no search term, with an empty string!

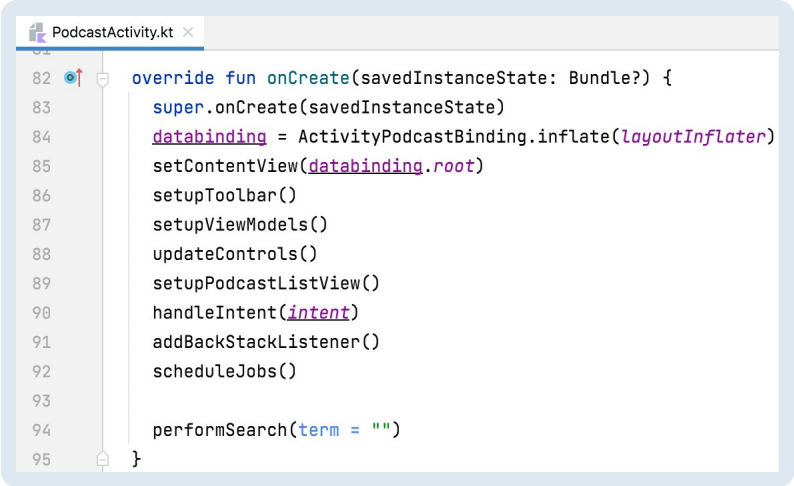
Scroll a bit down to find the response body.

```
2022-07-12 20:44:57.575 6740-6829/com.yourcompany.podplay I/okhttp.OkHttpClient:
2022-07-12 20:44:57.575 6740-6829/com.yourcompany.podplay I/okhttp.OkHttpClient: {
2022-07-12 20:44:57.575 6740-6829/com.yourcompany.podplay I/okhttp.OkHttpClient:  "resultCount":0,
2022-07-12 20:44:57.575 6740-6829/com.yourcompany.podplay I/okhttp.OkHttpClient:  "results": []
2022-07-12 20:44:57.575 6740-6829/com.yourcompany.podplay I/okhttp.OkHttpClient: }
```

Since there's no search term provided in the request query, you can see that the response also returned as an empty array instead of a proper response.

This certainly looks like a programming error, but it gives you a clue! Look for places where a network has been made with an empty input.

If you look back, this unusual network request was executed shortly after you launched the app. So look for `onCreate()` inside **PodcastActivity.kt**, which is the entry point when you launch PodPlay.



Voilà, there's a call for `performSearch(term = "")` - that seems to be responsible for this unusual network activity!

Remove `performSearch(term = "")` and launch the app again. Observe the network timeline and Logcat and notice there is no weird network call anymore.

Congratulations! You successfully eliminated the unnecessary network call.

Key Points

- Network Inspector offers a set of tools showing all the details available on network operations.
- Network timeline is the key to looking for network activity at any point since you launch the app.
- You can leverage Connection View and Thread View to look into different aspects of a network operation.
- Logging HTTP request and response data helps if you still can't find enough information through Network Inspector.

Where to Go From Here?

Your debugging skills have reached the next level as you've mastered the art of debugging network operations! To learn more on this topic, check out the tutorials and videos below:

- Android Networking: Fundamentals
- Securing Network Data
- Network Inspector Official Guide
- Android Studio Profiler

Don’t stop here! In the next chapter, you’ll learn to use **Energy Profiler** to optimize your app for minimizing resource consumption. Good luck on your next adventure!

12 Android Energy Profiler

Written by Zahidur Rahman Faisal

In the modern day, we can't live a single moment without our smartphones. Most of the time, we're either messaging, listening to music, watching videos or snapping pictures with our phones! To serve our needs all day long, smartphones are getting more and more powerful in terms of CPU, battery or storage. Even after having a smartphone with huge battery life, we all see the 'Low Battery' alert or run out of battery frequently. This indicates that no matter how powerful your phone is, the apps you use also need to smartly manage energy consumption so you can keep using them as long as you need.

In the previous chapter, you learned to profile network activity and optimize network resource consumption.

In this chapter, you'll learn to inspect energy consumption with the **Energy Profiler** that comes with Android Studio.

In the process, you'll learn about:

- How to use different components of the Energy Profiler tool.
- How to identify resource-hungry events, such as wake locks, jobs and alarms.
- Optimizing energy consumption and monitoring changes.

Overview of The Energy Profiler

The Energy Profiler is a part of Android Profiler tools in Android Studio 3.0 and onwards. The Android Profilers help you investigate different app performance aspects, such as CPU usage, network or radioactivity, GPS & sensor data. The Energy Profiler displays a visualization of how much energy each of these components uses. It also shows you occurrences of system events such as wake locks, alarms, jobs, and location requests, that impact energy consumption.

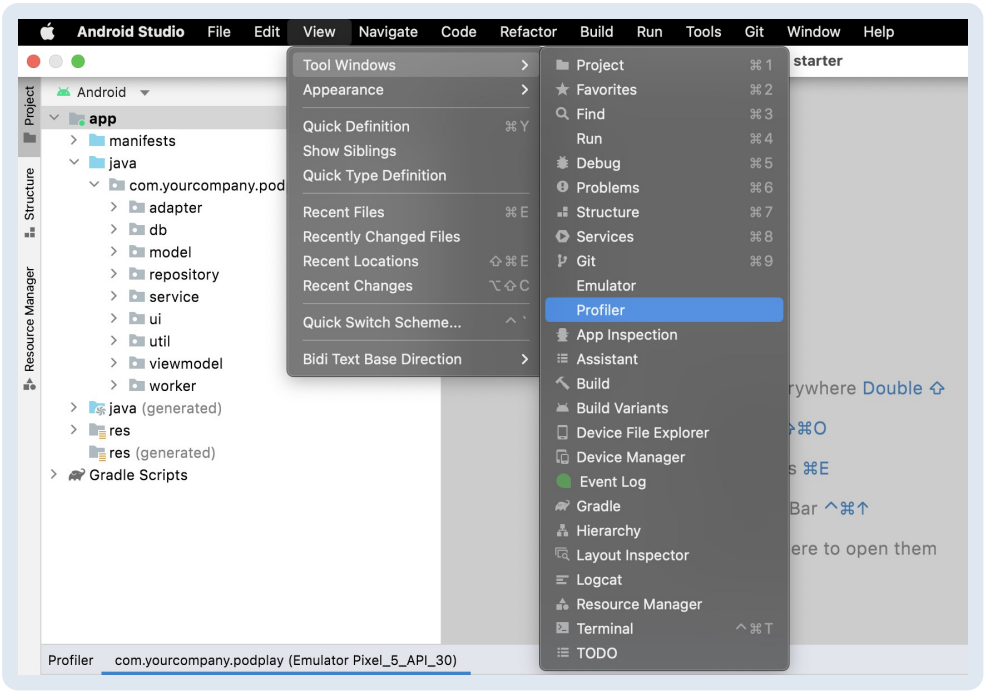
Energy Profiler appears as a row in the **Profiler** window once you run your app on a connected device or Android Emulator running Android 8.0 (API 26) or higher.

Follow these steps to open the Energy Profiler:

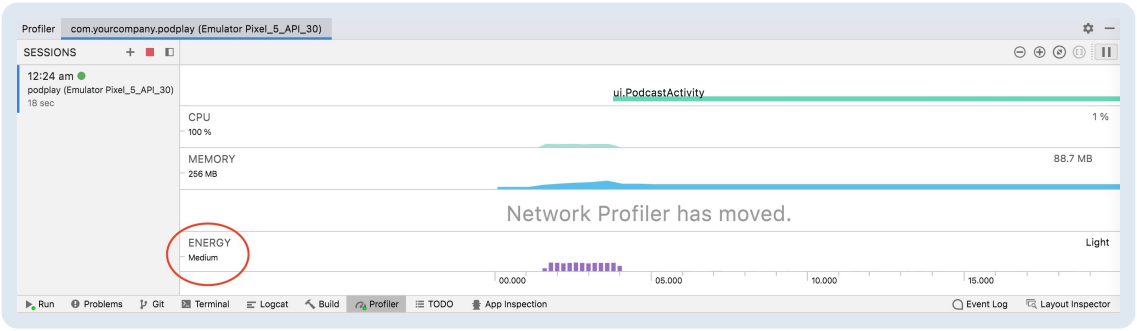
1. Open the **Podplay** starter project and run the app on an emulator or

connected device using API level 26 or higher.

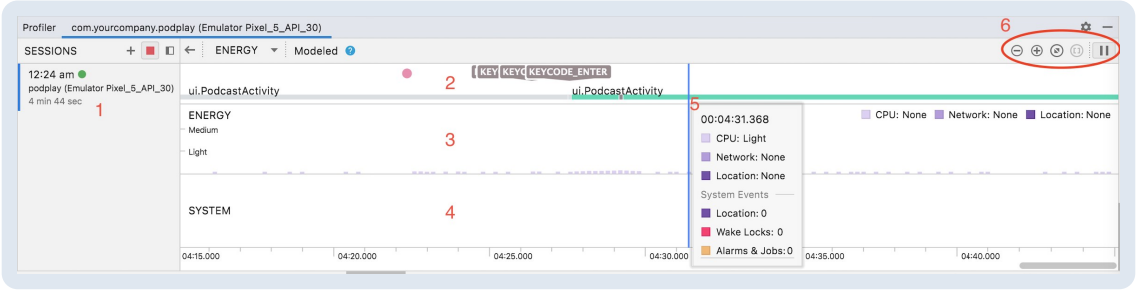
2. Select **View ▸ Tool Windows ▸ Profiler** from the menu bar.



3. This will start a new profiling session for PodPlay within the Android Profiler. Click over the **ENERGY** timeline from the right pane as displayed below:



This will open the actual Energy Profiler, and it's a detailed representation of the energy consumption from your app in real-time. The new panel will look like this:



As shown above, the Energy Profiler panel includes the following main components:

1. **Sessions Pane**: Displays your current session info, such as starting time,

elapsed time in seconds, and the device name. The sessions pane allows you to start a new profiling session or stop the current session. If you re-launch the app, a new session will be added to this pane with the current timestamp. The session data remains here until you quit Android Studio.

2. **Event Timeline**: This timeline shows user interactions with the device, such as keyboard inputs, pressing hardware buttons or screen rotation events.
3. **Energy Timeline**: This shows the estimated energy consumption from your application code. This includes CPU loads, Network activity etc.
4. **System Timeline**: System events that may affect energy consumption are displayed in this timeline. They could be alarms, scheduled jobs or wake locks. You'll get to more details about system events shortly.
5. **Quick Preview**: Using your cursor, hover over any point on the Energy Timeline. It'll display a quick preview of energy consumption from your app at that point.
6. **Timeline Controls**: These buttons offer you control over the timelines. The utility of them are as follows:

- **Zoom Out**: Zooms out of the timeline, and displays a longer period to allocate more requests, if there are any, in the timeline view.
- **Zoom In**: Zooms in on the timeline displaying a shorter period in the timeline view.
- **Reset Zoom**: Resets any zoom-out or zoom-in over the timeline view and reverts to the default zoom state.
- **Zoom to Selection**: This allows you to zoom into any selected interval from the timeline. This is helpful to focus on a specific time frame that you select to identify a network request which occurred within that period.
- **Attach to Live**: This button allows you to pause the timeline at any point or resume displaying the real-time updates jumping to the end of the timeline.

> **Note**: Selection over the timelines in the Energy Profiler *is not allowed*, so **Zoom to Selection** will remain disabled.

You might be curious to know more regarding energy management, for example, terms like **system events**, before putting those components in use!

Android Energy Management 101

Android runtime has a few techniques to ensure your device stays alive and performant for an extensive period. There are system events that consume

energy and defense mechanisms that protect the device from unnecessary energy drain. You need to clearly understand these to make your app energy-efficient.

System Events: Wake Locks, Jobs, and Alarms

Energy Profiler logs events that are triggered by the system that can affect energy consumption. These events can be a scheduled background task, a data request using sensors and more.

System events that are displayed in the Energy Profiler fall into the categories below:

- **Wake Locks**: There are cases when an app needs to keep the CPU or the screen awake to complete some work. For example, a video-player app might keep the screen on even when there's no user interaction for a while. A 'wake lock' is a system for keeping the CPU and screen on and in use for such cases. Otherwise, the device would go to sleep after a certain time to save energy.
- **Alarms or Background Tasks**: Using AlarmManager to schedule background tasks to be executed in the future or at regular intervals is a common approach in Android. However, it's not the best way to do so. When an alarm's triggered, it may wake up the device and run energy-consuming operations.
- **Location Requests**: Location requests made by GPS consume significant amounts of energy. That's one of the over-used features in smartphone apps and drains energy quickly.

Doze Mode

From Android 6.0 (API level 23) onwards, Android introduced two power-saving features that extend battery life by managing how apps behave in different conditions. One of them is putting the device on **Doze** mode when it's not connected to a power source, in simple words, charging.

Doze mode reduces battery consumption by halting background CPU and network activity when you're not using a device for an extended amount of time. If a user leaves a device unplugged and stationary for a while, the device enters Doze mode.

Below are the restrictions applied while the device is in Doze mode:

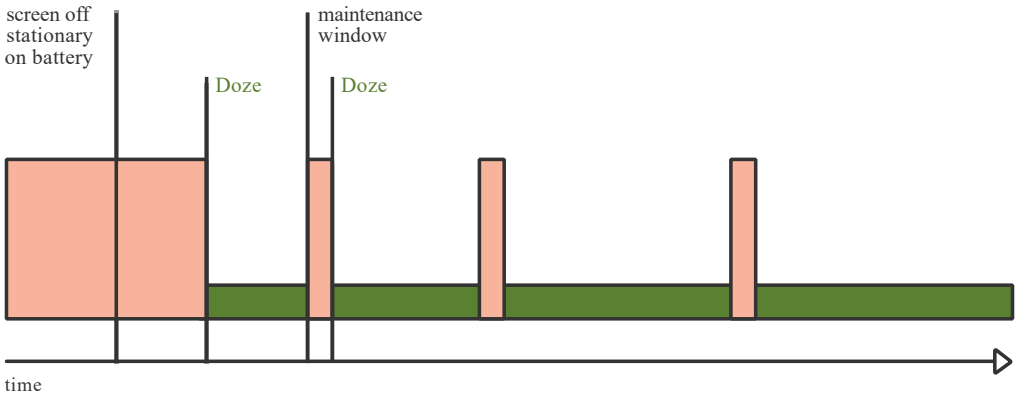
- Suspended network access.

- Ignores wake locks.
- Standard `AlarmManager` alarms are deferred.
- The system performs no Wi-Fi scans.
- Sync adapters aren't allowed to run.
- `JobScheduler`s are paused.

You must be thinking, how does the system resume these operations then?

Well, the system exits Doze for short intervals to let apps complete their deferred activities. During this time, the system runs all pending syncs, jobs, alarms and so on. This period is the **maintenance window**.

The diagram below makes it easy to visualize how Doze mode works:



At the end of each maintenance window the system enters Doze again, suspending activities mentioned earlier. The system schedules maintenance windows less frequently over time. This helps to reduce battery consumption in cases of longer-term inactivity when the device isn't connected to a power source.

As soon as the user uses the device by moving it, turning on the screen, or connecting a charger, the system exits Doze, and all apps return to normal behavior.

Doze mode especially affects `AlarmManager` alarms and timers-related activities because alarms in Android 5.1 (API level 22) or lower don't fire when the system is in Doze!

App Standby

App Standby is another energy-saving technique that's used to prevent unnecessary system events. It's a state where the system determines if an app is idle or not in use. When in the standby state, apps are deferred from

background or network activity.

An app goes to the standby state when the user doesn't interact with the app for a certain period, and none of the following scenarios occur:

- The user intentionally launches the app.
- The app has any foreground process, either as an activity or service or being used by another activity or foreground service.
- The app receives a notification or generates a local notification.
- The app is an active device admin app, such as a device policy controller. Device admin apps never enter the App Standby state because they must remain available to receive policy from a server at any time.

The system reverts apps from the standby state when the user launches the app again or plugs the device into a power source, allowing them to access the network or execute any pending jobs and syncs. If the device is idle for a long time, the system allows apps on standby to access the network about once a day.

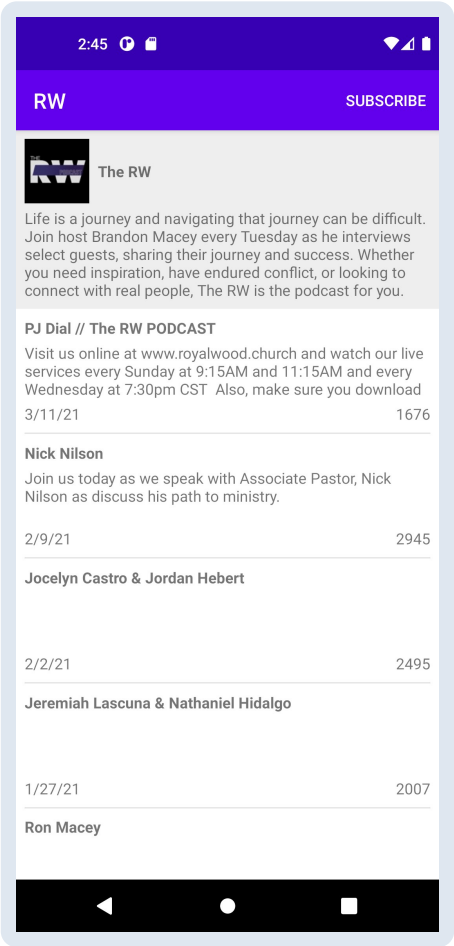
Inspecting and Optimizing Energy Consumption

You must be eager to get your hands on the Energy Profiler and see how you can minimize energy consumption from PodPlay!

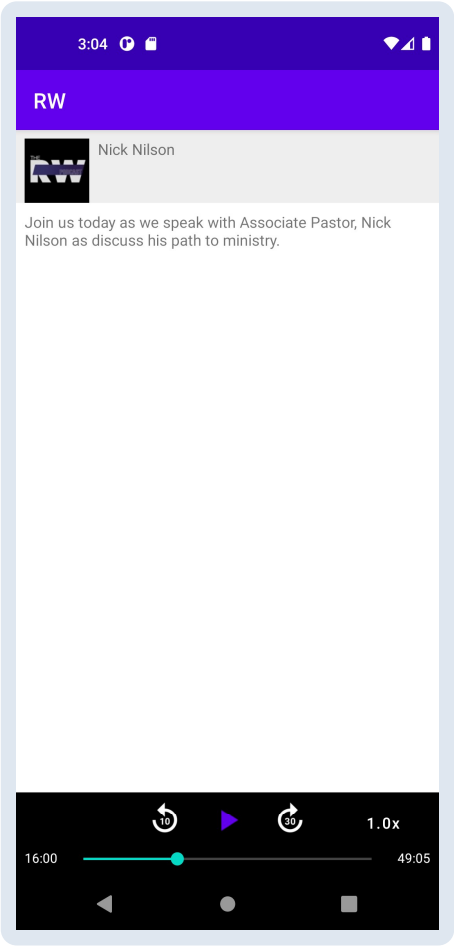
Run the app and go straight to the Energy Profiler by clicking the **ENERGY** timeline.

Observe the Energy and System timeline in this panel — all seems to be normal. Try performing some energy-intensive operations that the app's supposed to do, for example, playing podcasts, by following these steps:

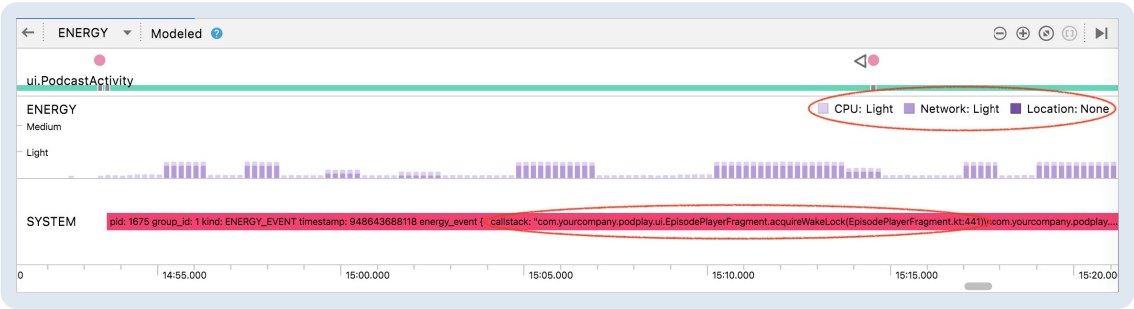
1. Search for the "RW" term and select **Return**.
2. From the **channel** list, select first. That'll open the channel details screen:



3. Start any podcast by tapping an item. The podcast player will launch. That's `EpisodePlayerFragment`:



Look at your Energy Profiler timeline. Now, you'll see some anomalies there:



The spikes in the **ENERGY** timeline indicate some resource-consuming activity. Looking at the highlighted area above, you can be relieved that the CPU and Network usage is still **Light**, which is a good sign.

But what's the long, red-colored bar in the **SYSTEM** timeline? That indicates something needs to be taken care of!

The System timeline displays a color-coded bar for the time range when a system event is active. Different color codes denote the different types of system events:

- **Red**: Wake locks.
- **Yellow**: Jobs and alarms.

- **Purple**: Location events.

If you look carefully at the wake lock, it clearly hints that the wake lock has been called from `EpisodePlayerFragment` at **line 441**.

Navigate to that line on **EpisodePlayerFragment.kt**. You'll see `acquireWakeLock()` initiating the wake lock `onStart()` of the `Fragment`.

As a media player app, PodPlay may use a wake lock to keep the CPU and screen awake while you're listening to the podcasts, but it's also important to release the wake lock as soon as the user leaves the player screen. That'll save energy consumption and prevent the display from being lit unnecessarily.

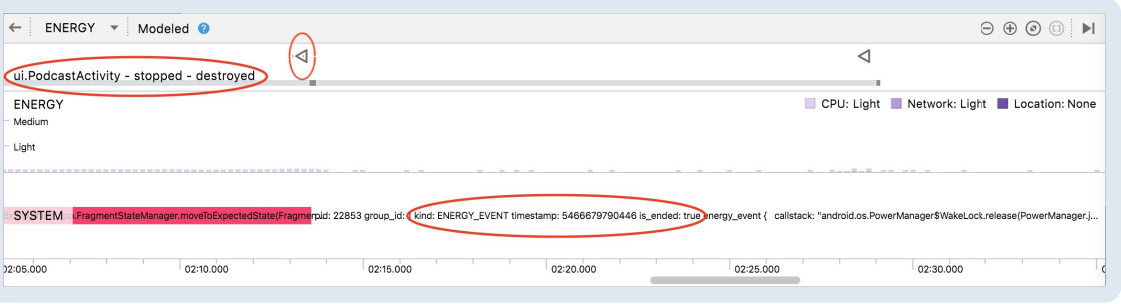
To do so, update the `onDestroyView()` block on **EpisodePlayerFragment.kt** as follows:

```
override fun onDestroyView() {
  super.onDestroyView()
  releaseWakeLock()
  _databinding = null
}
```

That'll release the wake lock whenever the user quits the player.

Now relaunch the app and play a podcast following the steps above. Then tap the **back** button until you see the **podcast list** screen again.

Observe the System timeline for a while. It'll update like below:



The highlights above confirm the energy event, wake lock, ended shortly after you tapped the back button to navigate away from the player screen. You can extract more info from the displayed logs, such as the **process ID (pid)**, **timestamp**, and **callstack**.

Your app is much more energy efficient now, congratulations!

You can play around a bit more or even try to remove the wake lock and see how that impacts the app through the Energy Profiler.

Best Practices for Energy Management

As a developer, you're in the ultimate control to code your app in a way that allows it to operate with minimal energy and resource consumption. Below are some best practices you can follow while developing your next app:

- Ensure wake locks are released as soon as they are no longer needed.
- If the app is performing any big HTTP downloads, such as media files, consider using DownloadManager.
- If the app needs to synchronize data periodically from a server, use a sync adapter.
- If the app requires background services, consider using WorkManager or JobScheduler instead of AlarmManager.
- Profile early and often to discover energy or performance issues to keep your app up to the mark!

Key Points

- Use the Energy Profiler to monitor, inspect and detect energy-related issues.
- Check out detailed energy stats at any point with the Energy and System events timeline.
- System events are resource-hungry! Be mindful while using them.
- Avoid triggering unnecessary network calls, background tasks or location requests to keep your app energy efficient.

Where to Go From Here?

That ends your adventure of diving as a debugger into the ocean of code and finding the finest solutions!

In this final chapter, you learned how to make the best use of the energy that your smartphone offers you. There's always more to learn, so continue your quest for energy with these advanced topics:

- Power Management
- Test power-related issues
- Profile battery usage with Batterystats and Battery Historian
- Optimize for Doze and App Standby
- Minimize the effect of regular updates

13 Conclusion

Congratulations on completing your debugging journey! It's been a long way — from learning the basics of Android debugging tools to handling app performance issues.

At this point, you have gained a lot of experience with debugging tools and techniques. But don't stop here! Hopefully, these techniques will inspire you to get more ideas about possible solutions. You'll also be faster in detecting issues and fixing them because now you know how to gather more information about the source of the problem.

One thing's for sure — with this knowledge, you'll move your app to the next level!

In case you're interested in more debugging content on our site, check out the Beginning Android Debugging video course about debugging basics, Android Debug Bridge (ADB): Beyond the Basics for an extended version of the ADB topic and Android Memory Profiler: Getting Started for learning about memory management.

If you have any questions or comments as you work through this book, please stop by our forums at https://forums.raywenderlich.com and look for the particular forum category for this book.

Thank you again for purchasing this book. Your continued support is what makes the books, tutorials, videos and other things we do at raywenderlich.com possible. We truly appreciate it!

– The *Android Debugging by Tutorials* team