Android Debug Bridge (adb)

Archived: 2026-04-05 23:48:27 UTC

Android Debug Bridge (`adb`) is a versatile command-line tool that lets you communicate with a device. The `adb` command facilitates a variety of device actions, such as installing and debugging apps. `adb` provides access to a Unix shell that you can use to run a variety of commands on a device. It is a client-server program that includes three components:

- **A client**, which sends commands. The client runs on your development machine. You can invoke a client from a command-line terminal by issuing an `adb` command.
- **A daemon (adbd)**, which runs commands on a device. The daemon runs as a background process on each device.
- **A server**, which manages communication between the client and the daemon. The server runs as a background process on your development machine.

`adb` is included in the Android SDK Platform Tools package. Download this package with the SDK Manager, which installs it at *`android_sdk`*`/platform-tools/` . If you want the standalone Android SDK Platform Tools package, download it here.

For information on connecting a device for use over `adb` , including how to use the Connection Assistant to troubleshoot common problems, see Run apps on a hardware device.

How adb works

When you start an `adb` client, the client first checks whether there is an `adb` server process already running. If there isn't, it starts the server process. When the server starts, it binds to local TCP port 5037 and listens for commands sent from `adb` clients.

Note: All `adb` clients use port 5037 to communicate with the `adb` server.

The server then sets up connections to all running devices. It locates emulators by scanning odd-numbered ports in the range 5555 to 5585, which is the range used by the first 16 emulators. Where the server finds an `adb` daemon (adbd), it sets up a connection to that port.

Each emulator uses a pair of sequential ports — an even-numbered port for console connections and an odd-numbered port for `adb` connections. For example:

Emulator 1, console: 5554
Emulator 1, `adb` : 5555
Emulator 2, console: 5556
Emulator 2, `adb` : 5557
and so on.

As shown, the emulator connected to `adb` on port 5555 is the same as the emulator whose console listens on port 5554.

Once the server has set up connections to all devices, you can use `adb` commands to access those devices. Because the server manages connections to devices and handles commands from multiple `adb` clients, you can control any device from any client or from a script.

Enable adb debugging on your device

To use adb with a device connected over USB, you must enable **USB debugging** in the device system settings, under **Developer options**. On Android 4.2 (API level 17) and higher, the **Developer options** screen is hidden by default. To make it visible, enable Developer options.

You can now connect your device with USB. You can verify that your device is connected by executing `adb devices` from the *android_sdk*`/platform-tools/` directory. If connected, you'll see the device name listed as a "device."

Note: When you connect a device running Android 4.2.2 (API level 17) or higher, the system shows a dialog asking whether to accept an RSA key that allows debugging through this computer. This security mechanism protects user devices because it ensures that USB debugging and other adb commands cannot be executed unless you're able to unlock the device and acknowledge the dialog.

For more information about connecting to a device over USB, read Run apps on a hardware device.

Connect to a device over Wi-Fi

Note: The instructions below do not apply to Wear devices running Android 11 (API level 30). See the guide to debugging a Wear OS app for more information.

Android 11 (API level 30) and higher support deploying and debugging your app wirelessly from your workstation using Android Debug Bridge (adb). For example, you can deploy your debuggable app to multiple remote devices without ever needing to physically connect your device via USB. This eliminates the need to deal with common USB connection issues, such as driver installation.

Before you begin using wireless debugging, do the following:

- Ensure that your workstation and device are connected to the same wireless network.
- Ensure that your device is running Android 11 (API level 30) or higher for phone or Android 13 (API level 33) or higher for TV and WearOS. For more information, see Check & update your Android version.
- If using the IDE, ensure that you have the latest version of Android Studio installed. You can download it here.
- On your workstation, update to the latest version of the SDK Platform Tools.

To use wireless debugging, you must pair your device to your workstation using a QR code or a pairing code. Your workstation and device must be connected to the same wireless network. To connect to your device, follow these steps:

1. Enable developer options on your device.

2. Open Android Studio and select **Pair Devices Using Wi-Fi** from the run configurations menu.

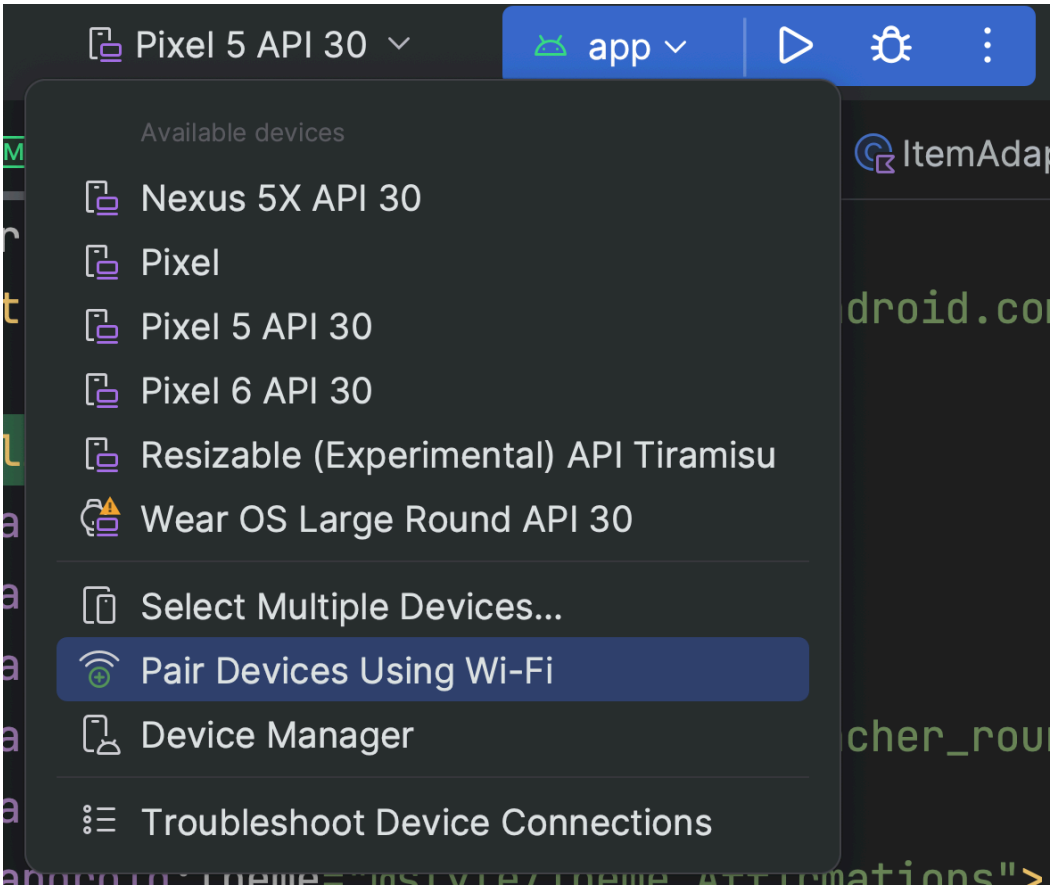

Figure 1. Run configurations menu.

The **Pair devices over Wi-Fi** window pops up, as shown in figure 2.

Figure 2. Popup window to pair devices using QR code or pairing code.

3. On your device, tap **Wireless debugging** and pair your device:

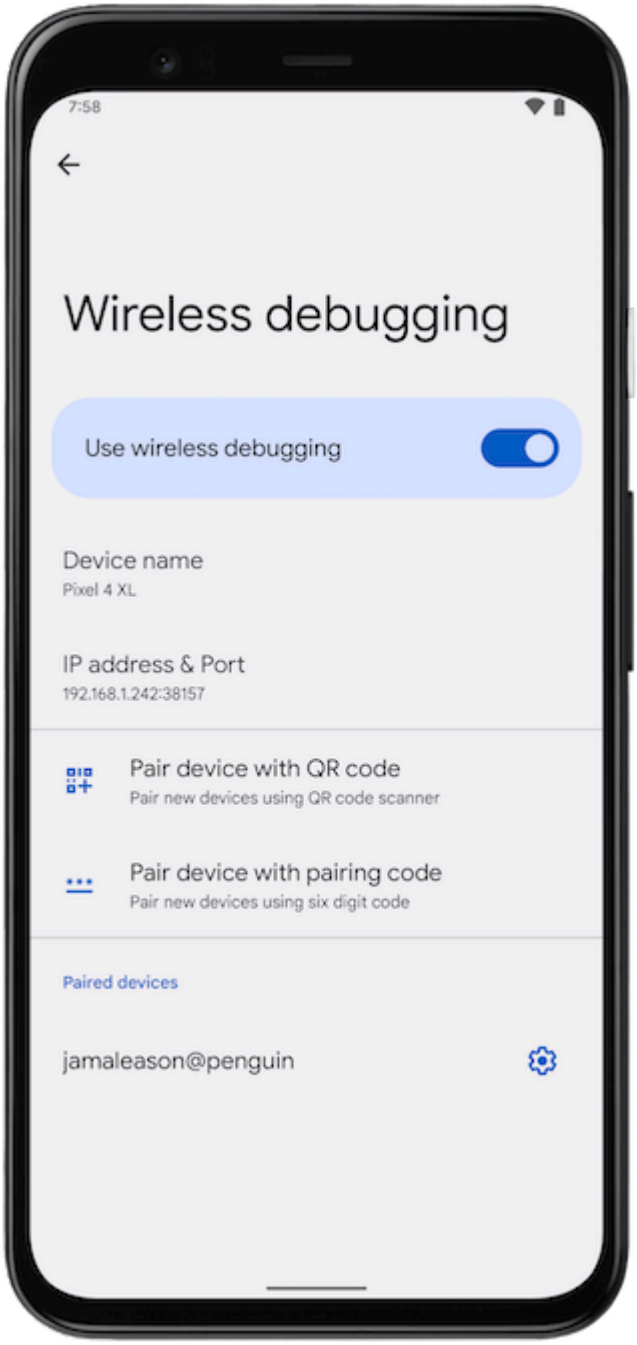

Figure 3. Screenshot of the **Wireless debugging** setting on a Google Pixel phone.

1. To pair your device with a QR code, select **Pair device with QR code** and scan the QR code obtained from the **Pair devices over Wi-Fi** popup shown in figure 2.

2. To pair your device with a pairing code, select **Pair device with pairing code** from the **Pair devices over Wi-Fi** popup. On your device, select **Pair using pairing code** and take note of the six-digit code provided. Once your device appears on the **Pair devices over Wi-Fi** window, you can select **Pair** and enter the six-digit code shown on your device.

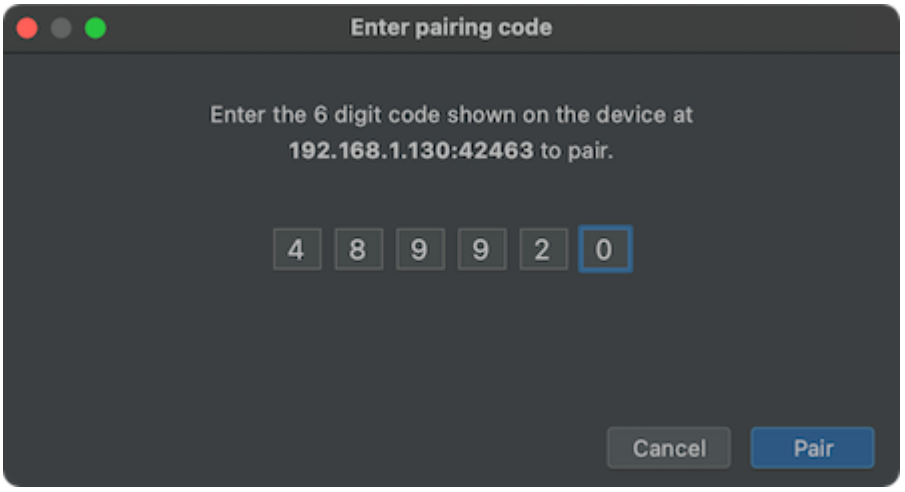

Figure 4. Example of six-digit code entry.

4. After your device is paired, you can attempt to deploy your app to your device.

   To pair a different device or to forget the current device on your workstation, navigate to **Wireless debugging** on your device. Tap your workstation name under **Paired devices** and select **Forget**.

5. If you want to quickly turn on and off wireless debugging, you can utilize the Quick settings developer tiles for **Wireless debugging**, found in **Developer Options** > **Quick settings developer tiles**.

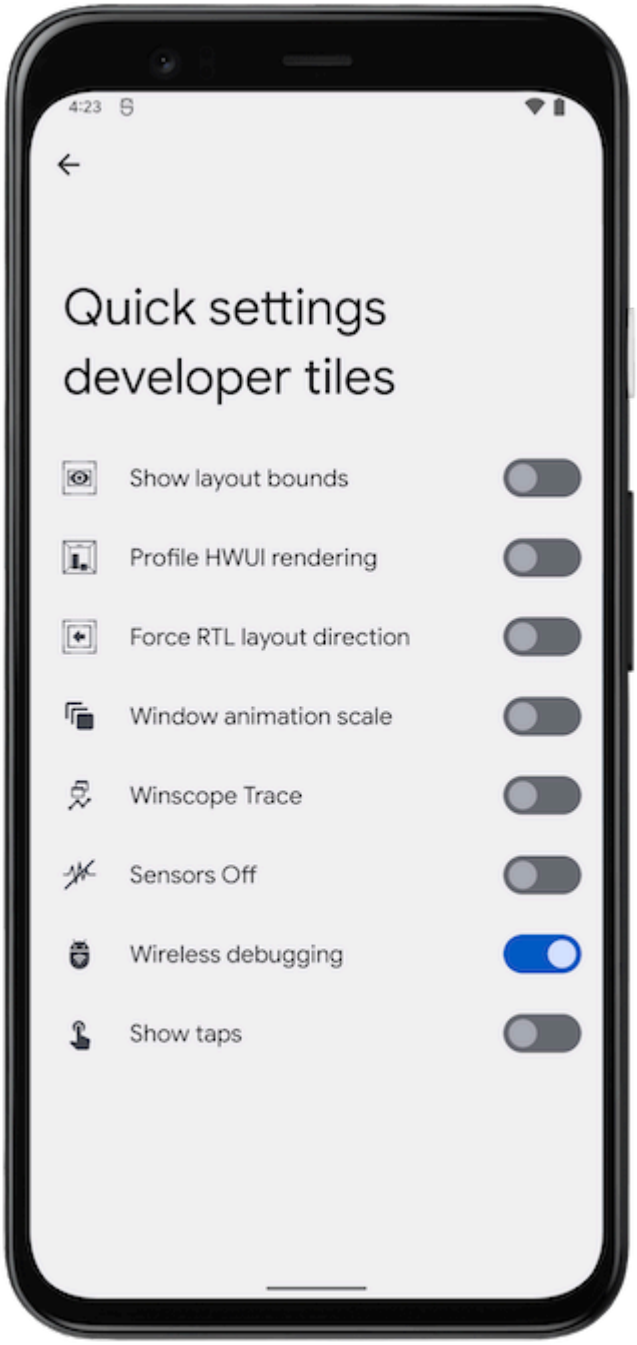

Figure 5. The **Quick settings developer tiles** setting lets you quickly turn wireless debugging on and off.

Wi-Fi connection using command line

Alternatively, to connect to your device using command line without Android Studio, follow these steps:

1. Enable developer options on your device, as described earlier.
2. Enable **Wireless debugging** on your device, as described earlier.
3. On your workstation, open a terminal window and navigate to `android_sdk/platform-tools`.

4. Find your IP address, port number, and pairing code by selecting **Pair device with pairing code**. Take note of the IP address, port number, and pairing code displayed on the device.
5. On your workstation's terminal, run `adb pair ipaddr:port`. Use the IP address and port number from above.
6. When prompted, enter the pairing code, as shown below.

```
Enter pairing code: 482924
Successfully paired to 192.168.1.130:37099 [guid=adb-235XY]
```

Figure 6. A message indicates that your device has been successfully paired.

Resolve wireless connection issues

If you are having issues connecting to your device wirelessly, try the following troubleshooting steps to resolve the issue.

Check whether your workstation and device meet the prerequisites

Check that the workstation and device meet the prerequisites listed at the beginning of this section.

Check for other known issues

The following is a list of current known issues with wireless debugging (with adb or Android Studio) and how to resolve them:

- **Wi-Fi is not connecting**: Secure Wi-Fi networks, such as corporate Wi-Fi networks, may block p2p connections and not let you connect over Wi-Fi. Try connecting with a cable or another (non-corp) Wi-Fi network. Wireless connection using `adb connect ip:port` over tcp/ip (following an initial USB connection) is another option, in case resorting to a non-corp network is an option.
- `adb` **over Wi-Fi sometimes turns off automatically**: This can happen if the device either switches Wi-Fi networks or disconnects from the network. To resolve, re-connect to the network.
- **Device not connecting after pairing successfully**: `adb` relies on mDNS to discover and automatically connect to paired devices. If your network or device configuration does not support mDNS or has disabled it, then you need to manually connect to the device using `adb connect ip:port`.

Connect wirelessly with a device after an initial USB connection (only option available on Android 10 and lower)

Note: This workflow is applicable also to Android 11 (and higher), the caveat being that it also involves an *initial* connection over physical USB.

Note: The following instructions do not apply to Wear devices running Android 10 (API level 29) or lower. See the guide about debugging a Wear OS app for more information.

`adb` usually communicates with the device over USB, but you can also use `adb` over Wi-Fi. To connect a device running Android 10 (API level 29) or lower, follow these initial steps over USB:

1. Connect your Android device and `adb` host computer to a common Wi-Fi network.

   Note: Beware that not all access points are suitable. You might need to use an access point whose firewall is configured properly to support `adb`.

2. Connect the device to the host computer with a USB cable.
3. Set the target device to listen for a TCP/IP connection on port 5555:

   ```
   adb tcpip 5555
   ```

4. Disconnect the USB cable from the target device.
5. Find the IP address of the Android device. For example, on a Nexus device, you can find the IP address at **Settings** > **About tablet** (or **About phone**) > **Status** > **IP address**.
6. Connect to the device by its IP address:

   ```
   adb connect device_ip_address:5555
   ```

7. Confirm that your host computer is connected to the target device:

   ```
   $ adb devices
   List of devices attached
   device_ip_address:5555 device
   ```

Your device is now connected to `adb`.

If the `adb` connection to your device is lost:

- Make sure that your host is still connected to the same Wi-Fi network as your Android device.
- Reconnect by executing the `adb connect` step again.
- If that doesn't work, reset your `adb` host:

  ```
  adb kill-server
  ```

  Then start over from the beginning.

Query for devices

Before issuing `adb` commands, it is helpful to know what device instances are connected to the `adb` server. Generate a list of attached devices using the `devices` command:

```
adb devices -l
```

In response, `adb` prints this status information for each device:

- **Serial number:** `adb` creates a string to uniquely identify the device by its port number. Here's an example serial number: `emulator-5554`
- **State:** The connection state of the device can be one of the following:
  - `offline` : The device is not connected to `adb` or is not responding.
  - `device` : The device is connected to the `adb` server. Note that this state does not imply that the Android system is fully booted and operational, because the device connects to `adb` while the system is still booting. After boot-up, this is the normal operational state of a device.
  - `no device` : There is no device connected.
- **Description:** If you include the `-l` option, the `devices` command tells you what the device is. This information is helpful when you have multiple devices connected so that you can tell them apart.

The following example shows the `devices` command and its output. There are three devices running. The first two lines in the list are emulators, and the third line is a hardware device that is attached to the computer.

```
$ adb devices
List of devices attached
emulator-5556 device product:sdk_google_phone_x86_64 model:Android_SDK_built_for_x86_64 device:gener
emulator-5554 device product:sdk_google_phone_x86 model:Android_SDK_built_for_x86 device:generic_x86
0a388e93      device usb:1-1 product:razor model:Nexus_7 device:flo
```

Emulator not listed

The `adb devices` command has a corner-case command sequence that causes running emulators to not show up in the `adb devices` output even though the emulators are visible on your desktop. This happens when *all* of the following conditions are true:

- The `adb` server is not running.
- You use the `emulator` command with the `-port` or `-ports` option with an odd-numbered port value between 5554 and 5584.
- The odd-numbered port you chose is not busy, so the port connection can be made at the specified port number — or, if it is busy, the emulator switches to another port that meets the requirements in 2.
- You start the `adb` server after you start the emulator.

One way to avoid this situation is to let the emulator choose its own ports and to run no more than 16 emulators at once. Another way is to always start the `adb` server before you use the `emulator` command, as explained in the following examples.

Example 1: In the following command sequence, the `adb devices` command starts the `adb` server, but the list of devices does not appear.

Stop the `adb` server and enter the following commands in the order shown. For the AVD name, provide a valid AVD name from your system. To get a list of AVD names, type `emulator -list-avds`. The `emulator` command is in the *`android_sdk`*`/tools` directory.

```
$ adb kill-server
$ emulator -avd Nexus_6_API_25 -port 5555
$ adb devices

List of devices attached
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
```

Example 2: In the following command sequence, `adb devices` displays the list of devices because the `adb` server was started first.

To see the emulator in the `adb devices` output, stop the `adb` server, and then start it again after using the `emulator` command and before using the `adb devices` command, as follows:

```
$ adb kill-server
$ emulator -avd Nexus_6_API_25 -port 5557
$ adb start-server
$ adb devices

List of devices attached
emulator-5557 device
```

For more information about emulator command-line options, see Command-Line startup options.

Send commands to a specific device

If multiple devices are running, you must specify the target device when you issue the `adb` command. To specify the target, follow these steps:

1. Use the `devices` command to get the serial number of the target.
2. Once you have the serial number, use the `-s` option with the `adb` commands to specify the serial number.
    1. If you're going to issue a lot of `adb` commands, you can set the `$ANDROID_SERIAL` environment variable to contain the serial number instead.
    2. If you use both `-s` and `$ANDROID_SERIAL`, `-s` overrides `$ANDROID_SERIAL`.

In the following example, the list of attached devices is obtained, and then the serial number of one of the devices is used to install the `helloWorld.apk` on that device:

```
$ adb devices
List of devices attached
```

```
emulator-5554 device
emulator-5555 device
0.0.0.0:6520  device

# To install on emulator-5555
$ adb -s emulator-5555 install helloWorld.apk
# To install on 0.0.0.0:6520
$ adb -s 0.0.0.0:6520 install helloWorld.apk
```

Note: If you issue a command without specifying a target device when multiple devices are available, `adb` displays an error "adb: more than one device/emulator".

If you have multiple devices available but only one is an emulator, use the `-e` option to send commands to the emulator. If there are multiple devices but only one hardware device attached, use the `-d` option to send commands to the hardware device.

Install an app

You can use `adb` to install an APK on an emulator or connected device with the `install` command:

```
adb install path_to_apk
```

You must use the `-t` option with the `install` command when you install a test APK. For more information, see `-t` .

To install multiple APKs use `install-multiple` . This is useful if you download all the APKs for a specific device for your app from the Play Console and want to install them on an emulator or physical device.

For more information about how to create an APK file that you can install on an emulator/device instance, see Build and run your app.

Note: If you are using Android Studio, you do not need to use `adb` directly to install your app on the emulator or device. Instead, Android Studio handles the packaging and installation of the app for you.

Set up port forwarding

Use the `forward` command to set up arbitrary port forwarding, which forwards requests on a specific host port to a different port on a device. The following example sets up forwarding of host port 6100 to device port 7100:

```
adb forward tcp:6100 tcp:7100
```

The following example sets up forwarding of host port 6100 to local:logd:

```
adb forward tcp:6100 local:logd
```

This could be useful if you are trying to detemine what is being sent to a given port on the device. All received data will be written to the system-logging daemon and displayed in the device logs.

Copy files to and from a device

Use the `pull` and `push` commands to copy files to and from a device. Unlike the `install` command, which only copies an APK file to a specific location, the `pull` and `push` commands let you copy arbitrary directories and files to any location in a device.

To copy a file or directory and its sub-directories *from* the device, do the following:

```
adb pull remote local
```

To copy a file or directory and its sub-directories *to* the device, do the following:

```
adb push local remote
```

Replace `local` and `remote` with the paths to the target files/directory on your development machine (local) and on the device (remote). For example:

```
adb push myfile.txt /sdcard/myfile.txt
```

Stop the adb server

In some cases, you might need to terminate the `adb` server process and then restart it to resolve the problem. For example, this could be the case if `adb` does not respond to a command.

To stop the `adb` server, use the `adb kill-server` command. You can then restart the server by issuing any other `adb` command.

Issue adb commands

Issue `adb` commands from a command line on your development machine or from a script using the following:

```
adb [-d | -e | -s serial_number] command
```

If there's only one emulator running or only one device connected, the `adb` command is sent to that device by default. If multiple emulators are running and/or multiple devices are attached, you need to use the `-d`, `-e`, or `-s` option to specify the target device to which the command should be directed.

You can see a detailed list of all supported `adb` commands using the following command:

```
adb --help
```

Issue shell commands

You can use the `shell` command to issue device commands through `adb` or to start an interactive shell. To issue a single command, use the `shell` command like this:

```
adb [-d |-e | -s serial_number] shell shell_command
```

To start an interactive shell on a device, use the `shell` command like this:

```
adb [-d | -e | -s serial_number] shell
```

To exit an interactive shell, press `Control+D` or type `exit`.

Android provides most of the usual Unix command-line tools. For a list of available tools, use the following command:

```
adb shell ls /system/bin
```

Help is available for most of the commands via the `--help` argument. Many of the shell commands are provided by toybox. General help applicable to all toybox commands is available via `toybox --help`.

With Android Platform Tools 23 and higher, `adb` handles arguments the same way that the `ssh(1)` command does. This change has fixed a lot of problems with command injection and makes it possible to safely execute commands that contain shell metacharacters, such as `adb install Let\'sGo.apk`. This change means that the interpretation of any command that contains shell metacharacters has also changed.

For example, `adb shell setprop key 'two words'` is now an error, because the quotes are swallowed by the local shell, and the device sees `adb shell setprop key two words`. To make the command work, quote twice, once for the local shell and once for the remote shell, as you do with `ssh(1)`. For example, `adb shell setprop key "'two words'"` works because the local shell takes the outer level of quoting and the device still sees the inner level of quoting: `setprop key 'two words'`. Escaping is also an option, but quoting twice is usually easier.

See also Logcat command-line tool, which is useful for monitoring the system log.

Call activity manager

Within an `adb` shell, you can issue commands with the activity manager (`am`) tool to perform various system actions, such as start an activity, force-stop a process, broadcast an intent, modify the device screen properties, and more.

While in a shell, the `am` syntax is:

```
am command
```

You can also issue an activity manager command directly from `adb` without entering a remote shell. For example:

```
adb shell am start -a android.intent.action.VIEW
```

Table 1. Available activity manager commands

Command	Description
`start [options] intent`	Start an `Activity` specified by *intent*. See the Specification for intent arguments. Options are: • `-D` : Enable debugging. • `-W` : Wait for launch to complete. • `--start-profiler file` : Start profiler and send results to *file*. • `-P file` : Like `--start-profiler`, but profiling stops when the app goes idle. • `-R count` : Repeat the activity launch *count* times. Prior to each repeat, the top activity will be finished. • `-S` : Force stop the target app before starting the activity. • `--opengl-trace` : Enable tracing of OpenGL functions. • `--user user_id | current` : Specify which user to run as; if not specified, then run as the current user.
`startservice [options] intent`	Start the `Service` specified by *intent*. See the Specification for intent arguments. Options are: • `--user user_id | current` : Specify which user to run as. If not specified, then run as the current user.
`force-stop package`	Force-stop everything associated with *package*.
`kill [options] package`	Kill all processes associated with *package*. This command kills only processes that are safe to kill and that will not impact the user experience. Options are:

	• `--user user_id | all | current` : Specify which user's processes to kill. If not specified, then kill all users' processes.
`kill-all`	Kill all background processes.
`broadcast [options] intent`	Issue a broadcast intent. See the Specification for intent arguments. Options are: • `[--user user_id | all | current]` : Specify which user to send to. If not specified, then send to all users.
`instrument [options] component`	Start monitoring with an `Instrumentation` instance. Typically the target `component` is the form `test_package/runner_class` . Options are: • `-r` : Print raw results (otherwise decode `report_key_streamresult`). Use with `[-e perf true]` to generate raw output for performance measurements. • `-e name value` : Set argument `name` to `value` . For test runners a common form is `-e testrunner_flag value[,value...]` . • `-p file` : Write profiling data to `file` . • `-w` : Wait for instrumentation to finish before returning. Required for test runners. • `--no-window-animation` : Turn off window animations while running. • `--user user_id | current` : Specify which user instrumentation runs in. If not specified, run in the current user.
`profile start process file`	Start profiler on `process` , write results to `file` .
`profile stop process`	Stop profiler on `process` .
`dumpheap [options] process file`	Dump the heap of `process` , write to `file` . Options are: • `--user [user_id | current]` : When supplying a process name, specify the user of the process to dump. If not specified, the current user is used. • `-b [| png | jpg | webp]` : Dump bitmaps from graphics memory (API level 35 and above). Optionally specify the format to dump in (PNG by

	default). • `-n` : Dump native heap instead of managed heap.
`dumpbitmaps [*options*] [-p *process*]`	Dump bitmap information from `*process*` (API level 36 and above). Options are: • `-d|--dump [*format*]` : dump bitmap contents in the specified `*format*`, which can be one of `png`, `jpg`, or `webp`, default to `png` if none is specified. A zip file `dumpbitmaps-<time>.zip` will be created with the bitmaps. • `-p *process*` : dump bitmaps from `*process*`, multiple `-p *process*` can be specified. If no `*process*` is specified, bitmaps from all processes will be dumped.
`set-debug-app [*options*] *package*`	Set app `*package*` to debug. Options are: • `-w` : Wait for debugger when app starts. • `--persistent` : Retain this value.
`clear-debug-app`	Clear the package previous set for debugging with `set-debug-app`.
`monitor [*options*]`	Start monitoring for crashes or ANRs. Options are: • `--gdb` : Start `gdbserv` on the given port at crash/ANR.
`screen-compat {on | off} *package*`	Control screen compatibility mode of `*package*`.
`display-size [reset | *width*x*height*]`	Override device display size. This command is helpful for testing your app across different screen sizes by mimicking a small screen resolution using a device with a large screen, and vice versa. Example: `am display-size 1280x800`
`display-density *dpi*`	Override device display density. This command is helpful for testing your app across different screen densities by mimicking a high-density screen environment using a low-density screen, and vice versa.

	Example: `am display-density 480`
`to-uri` *`intent`*	Print the given intent specification as a URI. See the Specification for intent arguments.
`to-intent-uri` *`intent`*	Print the given intent specification as an `intent:` URI. See the Specification for intent arguments.

Specification for intent arguments

For activity manager commands that take an *`intent`* argument, you can specify the intent with the following options:

Show all

`-a` *`action`*
: Specify the intent action, such as `android.intent.action.VIEW`. You can declare this only once.

`-d` *`data_uri`*
: Specify the intent data URI, such as `content://contacts/people/1`. You can declare this only once.

`-t` *`mime_type`*
: Specify the intent MIME type, such as `image/png`. You can declare this only once.

`-c` *`category`*
: Specify an intent category, such as `android.intent.category.APP_CONTACTS`.

`-n` *`component`*
: Specify the component name with package name prefix to create an explicit intent, such as `com.example.app/.ExampleActivity`.

`-f` *`flags`*
: Add flags to the intent, as supported by `setFlags()`.

`--esn` *`extra_key`*
: Add a null extra. This option is not supported for URI intents.

`-e | --es` *`extra_key extra_string_value`*
: Add string data as a key-value pair.

`--ez` *`extra_key extra_boolean_value`*
: Add boolean data as a key-value pair.

`--ei` *`extra_key extra_int_value`*
: Add integer data as a key-value pair.

`--el` *`extra_key extra_long_value`*
: Add long data as a key-value pair.

`--ef` *`extra_key extra_float_value`*
: Add float data as a key-value pair.

`--eu extra_key extra_uri_value`
: Add URI data as a key-value pair.

`--ecn extra_key extra_component_name_value`
: Add a component name, which is converted and passed as a `ComponentName` object.

`--eia extra_key extra_int_value[,extra_int_value...]`
: Add an array of integers.

`--ela extra_key extra_long_value[,extra_long_value...]`
: Add an array of longs.

`--efa extra_key extra_float_value[,extra_float_value...]`
: Add an array of floats.

`--grant-read-uri-permission`
: Include the flag `FLAG_GRANT_READ_URI_PERMISSION`.

`--grant-write-uri-permission`
: Include the flag `FLAG_GRANT_WRITE_URI_PERMISSION`.

`--debug-log-resolution`
: Include the flag `FLAG_DEBUG_LOG_RESOLUTION`.

`--exclude-stopped-packages`
: Include the flag `FLAG_EXCLUDE_STOPPED_PACKAGES`.

`--include-stopped-packages`
: Include the flag `FLAG_INCLUDE_STOPPED_PACKAGES`.

`--activity-brought-to-front`
: Include the flag `FLAG_ACTIVITY_BROUGHT_TO_FRONT`.

`--activity-clear-top`
: Include the flag `FLAG_ACTIVITY_CLEAR_TOP`.

`--activity-clear-when-task-reset`
: Include the flag `FLAG_ACTIVITY_CLEAR_WHEN_TASK_RESET`.

`--activity-exclude-from-recents`
: Include the flag `FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS`.

`--activity-launched-from-history`
: Include the flag `FLAG_ACTIVITY_LAUNCHED_FROM_HISTORY`.

`--activity-multiple-task`
: Include the flag `FLAG_ACTIVITY_MULTIPLE_TASK`.

`--activity-no-animation`
: Include the flag `FLAG_ACTIVITY_NO_ANIMATION`.

`--activity-no-history`
: Include the flag `FLAG_ACTIVITY_NO_HISTORY`.

`--activity-no-user-action`
: Include the flag `FLAG_ACTIVITY_NO_USER_ACTION`.

`--activity-previous-is-top`
: Include the flag `FLAG_ACTIVITY_PREVIOUS_IS_TOP`.

`--activity-reorder-to-front`
: Include the flag `FLAG_ACTIVITY_REORDER_TO_FRONT`.

`--activity-reset-task-if-needed`
: Include the flag `FLAG_ACTIVITY_RESET_TASK_IF_NEEDED`.

`--activity-single-top`
: Include the flag `FLAG_ACTIVITY_SINGLE_TOP`.

`--activity-clear-task`
: Include the flag `FLAG_ACTIVITY_CLEAR_TASK`.

`--activity-task-on-home`
: Include the flag `FLAG_ACTIVITY_TASK_ON_HOME`.

`--receiver-registered-only`
: Include the flag `FLAG_RECEIVER_REGISTERED_ONLY`.

`--receiver-replace-pending`
: Include the flag `FLAG_RECEIVER_REPLACE_PENDING`.

`--selector`
: Requires the use of `-d` and `-t` options to set the intent data and type.

`URI component package`
: You can directly specify a URI, package name, and component name when not qualified by one of the preceding options. When an argument is unqualified, the tool assumes the argument is a URI if it contains a ":" (colon). The tools assumes the argument is a component name if it contains a "/" (forward-slash); otherwise it assumes the argument is a package name.

Call package manager (`pm`)

Within an `adb` shell, you can issue commands with the package manager (`pm`) tool to perform actions and queries on app packages installed on the device.

While in a shell, the `pm` syntax is:

```
pm command
```

You can also issue a package manager command directly from `adb` without entering a remote shell. For example:

```
adb shell pm uninstall  com.example.MyApp
```

Table 2. Available package manager commands

Command	Description
`list packages [options] filter`	Print all packages, optionally only those whose package name contains the text in `filter`. Options: • `-f` : See associated file. • `-d` : Filter to only show disabled packages.

	• `-e` : Filter to only show enabled packages. • `-s` : Filter to only show system packages. • `-3` : Filter to only show third-party packages. • `-i` : See the installer for the packages. • `-u` : Include uninstalled packages. • `--user user_id` : The user space to query.
`list permission-groups`	Print all known permission groups.
`list permissions [options] group`	Print all known permissions, optionally only those in `group`. Options: • `-g` : Organize by group. • `-f` : Print all information. • `-s` : Short summary. • `-d` : Only list dangerous permissions. • `-u` : List only the permissions users will see.
`list instrumentation [options]`	List all test packages. Options: • `-f` : List the APK file for the test package. • `target_package` : List test packages for only this app.
`list features`	Print all features of the system.
`list libraries`	Print all the libraries supported by the current device.
`list users`	Print all users on the system.
`path package`	Print the path to the APK of the given `package`.
`install [options] path`	Install a package, specified by `path`, to the system. Options: • `-r` : Reinstall an existing app, keeping its data. • `-t` : Allow test APKs to be installed. Gradle generates a test APK when you have only run or debugged your app or have used the Android Studio **Build** > **Build APK** command. If the APK is built using a developer preview SDK, you must include the `-t` option with the `install` command if you are installing a test APK. • `-i installer_package_name` : Specify the installer package name.

	• `--user user_id` : Specify the user to install the package for. By default, the package is installed for all users existing on the device.
	• `--install-location location` : Set the install location using one of the following values:
	  ▪ `0` : Use the default install location.
	  ▪ `1` : Install on internal device storage.
	  ▪ `2` : Install on external media.
	• `-f` : Install package on the internal system memory.
	• `-d` : Allow version code downgrade.
	• `-g` : Grant all permissions listed in the app manifest.
	• `--fastdeploy` : Quickly update an installed package by only updating the parts of the APK that changed.
	• `--incremental` : Installs enough of the APK to launch the app while streaming the remaining data in the background. To use this feature, you must sign the APK, create an APK Signature Scheme v4 file, and place this file in the same directory as the APK. This feature is only supported on certain devices. This option forces `adb` to use the feature or fail if it is not supported, with verbose information on why it failed. Append the `--wait` option to wait until the APK is fully installed before granting access to the APK.
	
	`--no-incremental` prevents `adb` from using this feature.

`uninstall [options] package`	Removes a package from the system. Options: • `-k` : Keep the data and cache directories after package removal. • `--user user_id` : Specifies the user for whom the package is removed. By default, the package is removed for all users on the device. • `--versionCode version_code` : Only uninstalls if the app has the given version code.
`clear package`	Delete all data associated with a package.
`enable package_or_component`	Enable the given package or component (written as "package/class").
`disable package_or_component`	Disable the given package or component (written as "package/class").
`disable-user [options] package_or_component`	Options:

	• `--user user_id` : The user to disable.
`grant package_name permission`	Grant a permission to an app. On devices running Android 6.0 (API level 23) and higher, the permission can be any permission declared in the app manifest. On devices running Android 5.1 (API level 22) and lower, must be an optional permission defined by the app.
`revoke package_name permission`	Revoke a permission from an app. On devices running Android 6.0 (API level 23) and higher, the permission can be any permission declared in the app manifest. On devices running Android 5.1 (API level 22) and lower, must be an optional permission defined by the app.
`set-install-location location`	Change the default install location. Location values: • `0` : Auto: Let system decide the best location. • `1` : Internal: Install on internal device storage. • `2` : External: Install on external media. **Note:** This is only intended for debugging. Using this can cause apps to break and other undesireable behavior.
`get-install-location`	Returns the current install location. Return values: • `0 [auto]` : Let system decide the best location • `1 [internal]` : Install on internal device storage • `2 [external]` : Install on external media
`set-permission-enforced permission [true | false]`	Specify whether the given permission should be enforced.
`trim-caches desired_free_space`	Trim cache files to reach the given free space.
`create-user user_name`	Create a new user with the given `user_name` , printing the new user identifier of the user.
`remove-user user_id`	Remove the user with the given `user_id` , deleting all data associated with that user
`get-max-users`	Print the maximum number of users supported by the device.
`get-app-links [options] [package]`	Print the domain verification state for the given *package*, or for all packages if none is specified. State codes are defined as follows: • `none` : nothing has been recorded for this domain

	• `verified` : the domain has been successfully verified • `approved` : force-approved, usually through shell • `denied` : force-denied, usually through shell • `migrated` : preserved verification from a legacy response • `restored` : preserved verification from a user data restore • `legacy_failure` : rejected by a legacy verifier, unknown reason • `system_configured` : automatically approved by the device config • `>= 1024` : custom error code, which is specific to the device verifier Options are: • `--user user_id` : include user selections. Include all domains, not just autoVerify ones.
`reset-app-links [options] [package]`	Reset domain verification state for the given package, or for all packages if none is specified. • `package` : the package to reset, or "all" to reset all packages Options are: • `--user user_id` : include user selections. Include all domains, not just autoVerify ones.
`verify-app-links [--re-verify] [package]`	Broadcast a verification request for the given *package*, or for all packages if none is specified. Only sends if the package has previously not recorded a response. • `--re-verify` : send even if the package has recorded a response

`set-app-links [--package package] state domains`	Manually set the state of a domain for a package. The domain must be declared by the package as autoVerify for this to work. This command will not report a failure for domains that could not be applied.
	
	• `--package package` : the package to set, or "all" to set all packages
	• `state` : the code to set the domains to. Valid values are:
	  ◦ `STATE_NO_RESPONSE (0)` : reset as if no response was ever recorded.
	  ◦ `STATE_SUCCESS (1)` : treat domain as successfully verified by domain verification agent. Note that the domain verification agent can override this.

	◦ `STATE_APPROVED (2)` : treat domain as always approved, preventing the domain verification agent from changing it. ◦ `STATE_DENIED (3)` : treat domain as always denied, preventing the domain verification agent from changing it. • *`domains`* : space-separated list of domains to change, or "all" to change every domain.
`set-app-links-user-selection --user` *`user_id`* `[--package` *`package`*`]` *`enabled domains`*	Manually set the state of a host user selection for a package. The domain must be declared by the package for this to work. This command will not report a failure for domains that could not be applied. • `--user` *`user_id`* : the user to change selections for • `--package` *`package`* : the package to set • *`enabled`* : whether to approve the domain • *`domains`* : space-separated list of domains to change, or "all" to change every domain
`set-app-links-allowed --user` *`user_id`* `[--package` *`package`*`]` *`allowed`*	Toggle the auto-verified link-handling setting for a package. • `--user` *`user_id`* : the user to change selections for • `--package` *`package`* : the package to set, or "all" to set all packages; packages will be reset if no package is specified • *`allowed`* : true to allow the package to open auto-verified links, false to disable
`get-app-link-owners --user` *`user_id`* `[--package` *`package`*`]` *`domains`*	Print the owners for a specific domain for a given user in low- to high-priority order. • `--user` *`user_id`* : the user to query for • `--package` *`package`* : optionally also print for all web domains declared by a package, or "all" to print all packages • *`domains`* : space-separated list of domains to query for

Call device policy manager (`dpm`)

To help you develop and test your device management apps, issue commands to the device policy manager (`dpm`) tool. Use the tool to control the active admin app or change a policy's status data on the device.

While in a shell, the `dpm` syntax is:

```
dpm command
```

You can also issue a device policy manager command directly from `adb` without entering a remote shell:

```
adb shell dpm command
```

Table 3. Available device policy manager commands

Command	Description
`set-active-admin [options] component`	Sets *component* as active admin. Options are: • `--user user_id` : Specify the target user. You can also pass `--user current` to select the current user.
`set-profile-owner [options] component`	Set *component* as active admin and its package as profile owner for an existing user. Options are: • `--user user_id` : Specify the target user. You can also pass `--user current` to select the current user. • `--name name` : Specify the human-readable organization name.
`set-device-owner [options] component`	Set *component* as active admin and its package as device owner. Options are: • `--user user_id` : Specify the target user. You can also pass `--user current` to select the current user. • `--name name` : Specify the human-readable organization name.
`remove-active-admin [options] component`	Disable an active admin. The app must declare `android:testOnly` in the manifest. This command also removes device and profile owners. Options are: • `--user user_id` : Specify the target user. You can also pass `--user current` to select the current user.
`clear-freeze-period-record`	Clear the device's record of previously set freeze periods for system OTA updates. This is useful to avoid the device scheduling restrictions when developing apps that manage freeze periods. See Manage system updates.

	Supported on devices running Android 9.0 (API level 28) and higher.
`force-network-logs`	Force the system to make any existing network logs ready for retrieval by a DPC. If there are connection or DNS logs available, the DPC receives the `onNetworkLogsAvailable()` callback. See Network activity logging. This command is rate-limited. Supported on devices running Android 9.0 (API level 28) and higher.
`force-security-logs`	Force the system to make any existing security logs available to the DPC. If there are logs available, the DPC receives the `onSecurityLogsAvailable()` callback. See Log enterprise device activity. This command is rate-limited. Supported on devices running Android 9.0 (API level 28) and higher.

Take a screenshot

The `screencap` command is a shell utility for taking a screenshot of a device display.

While in a shell, the `screencap` syntax is:

```
screencap filename
```

To use `screencap` from the command line, enter the following:

```
adb shell screencap /sdcard/screen.png
```

Here's an example screenshot session, using the `adb` shell to capture the screenshot and the `pull` command to download the file from the device:

```
$ adb shell
shell@ $ screencap /sdcard/screen.png
shell@ $ exit
$ adb pull /sdcard/screen.png
```

Alternatively, if you omit the filename, `screencap` writes the image to standard output. When combined with the `-p` option to specify PNG format, you can stream the device screenshot directly to a file on your local machine.

Here's an example of capturing a screenshot and saving it locally in a single command:

```
# use 'exec-out' instead of 'shell' to get raw data
```

```
$ adb exec-out screencap -p > screen.png
```

Record a video

The `screenrecord` command is a shell utility for recording the display of devices running Android 4.4 (API level 19) and higher. The utility records screen activity to an MPEG-4 file. You can use this file to create promotional or training videos or for debugging and testing.

In a shell, use the following syntax:

```
screenrecord [options] filename
```

To use `screenrecord` from the command line, enter the following:

```
adb shell screenrecord /sdcard/demo.mp4
```

Stop the screen recording by pressing Control+C. Otherwise, the recording stops automatically at three minutes or the time limit set by `--time-limit`.

To begin recording your device screen, run the `screenrecord` command to record the video. Then, run the `pull` command to download the video from the device to the host computer. Here's an example recording session:

```
$ adb shell
shell@ $ screenrecord --verbose /sdcard/demo.mp4
(press Control + C to stop)
shell@ $ exit
$ adb pull /sdcard/demo.mp4
```

The `screenrecord` utility can record at any supported resolution and bit rate you request, while retaining the aspect ratio of the device display. The utility records at the native display resolution and orientation by default, with a maximum length of three minutes.

Limitations of the `screenrecord` utility:

- Audio is not recorded with the video file.
- Video recording is not available for devices running Wear OS.
- Some devices might not be able to record at their native display resolution. If you encounter problems with screen recording, try using a lower screen resolution.
- Rotation of the screen during recording is not supported. If the screen does rotate during recording, some of the screen is cut off in the recording.

Table 4. `screenrecord` options

Options	Description

`--help`	Display command syntax and options
`--size` *`widthxheight`*	Set the video size: `1280x720`. The default value is the device's native display resolution (if supported), 1280x720 if not. For best results, use a size supported by your device's Advanced Video Coding (AVC) encoder.
`--bit-rate` *`rate`*	Set the video bit rate for the video, in megabits per second. The default value is 20Mbps. You can increase the bit rate to improve video quality, but doing so results in larger movie files. The following example sets the recording bit rate to 6Mbps: `screenrecord --bit-rate 6000000 /sdcard/demo.mp4`
`--time-limit` *`time`*	Set the maximum recording time, in seconds. The default and maximum value is 180 (3 minutes).
`--rotate`	Rotate the output 90 degrees. This feature is experimental.
`--verbose`	Display log information on the command-line screen. If you do not set this option, the utility does not display any information while running.

Read ART profiles for apps

Starting in Android 7.0 (API level 24), the Android Runtime (ART) collects execution profiles for installed apps, which are used to optimize app performance. Examine the collected profiles to understand which methods are executed frequently and which classes are used during app startup.

Note: It is only possible to retrieve the execution profile filename if you have root access to the file system, for example, on an emulator.

To produce a text form of the profile information, use the following command:

```
adb shell cmd package dump-profiles package
```

To retrieve the file produced, use:

```
adb pull /data/misc/profman/package.prof.txt
```

Reset test devices

If you test your app across multiple test devices, it may be useful to reset your device between tests, for example, to remove user data and reset the test environment. You can perform a factory reset of a test device running Android 10 (API level 29) or higher using the `testharness` `adb` shell command, as shown:

```
adb shell cmd testharness enable
```

When restoring the device using `testharness`, the device automatically backs up the RSA key that allows debugging through the current workstation in a persistent location. That is, after the device is reset, the workstation can continue to debug and issue `adb` commands to the device without manually registering a new key.

Additionally, to help make it easier and more secure to keep testing your app, using the `testharness` to restore a device also changes the following device settings:

- The device sets up certain system settings so that initial device setup wizards do not appear. That is, the device enters a state from which you can quickly install, debug, and test your app.
- Settings:
  - Disables lock screen.
  - Disables emergency alerts.
  - Disables auto-sync for accounts.
  - Disables automatic system updates.
- Other:
  - Disables preinstalled security apps.

If your app needs to detect and adapt to the default settings of the `testharness` command, use the `ActivityManager.isRunningInUserTestHarness()`.

sqlite

`sqlite3` starts the `sqlite` command-line program for examining SQLite databases. It includes commands such as `.dump` to print the contents of a table and `.schema` to print the `SQL CREATE` statement for an existing table. You can also execute SQLite commands from the command line, as shown:

```
$ adb -s emulator-5554 shell
$ sqlite3 /data/data/com.example.app/databases/rssitems.db
SQLite version 3.3.12
Enter ".help" for instructions
```

Note: It is only possible to access a SQLite database if you have root access to the file system, for example, on an emulator.

For more information, see the `sqlite3` command line documentation.

adb USB backends

The adb server can interact with the USB stack through two backends. It can either use the native backend of the OS (Windows, Linux, or macOS) or it can use the `libusb` backend. Some features, such as `attach`, `detach`, and USB speed detection, are only available when using `libusb` backend.

You can choose a backend by using the `ADB_LIBUSB` environment variable. If it isn't set, adb uses its default backend. The default behavior varies among OS. Starting with ADB v34, the `liubusb` backend is used by default on all OS except Windows, where the native backend is used by default. If `ADB_LIBUSB` is set, it determines whether the native backend or `libusb` is used. See the adb manual page for more information about adb environment variables.

adb mDNS backends

ADB can use the multicast DNS protocol to automatically connect the server and devices. The ADB server ships with two backends, Bonjour (Apple's mdnsResponder) and Openscreen.

The Bonjour backend needs a daemon to be running on the host machine. On macOS Apple's built-in daemon is always running, but on Windows and Linux, the user must make sure the `mdnsd` daemon is up and running. If the command `adb mdns check` returns an error, it is likely that ADB is using the Bonjour backend but there is no Bonjour daemon running.

The Openscreen backend does not need a daemon to be running on the machine. Support for the Openscreen backend on macOS starts at ADB v35. Windows and Linux are supported as of ADB v34.

By default ADB uses the Bonjour backend. This behavior can be changed using the environment variable `ADB_MDNS_OPENSCREEN` (set to `1` or `0`). See the ADB manual page for further details.

adb Burst Mode (starting with ADB 36.0.0)

Burst Mode is an experimental feature that lets ADB to keep on sending packets to a device even before the device has responded to the previous packet. This greatly increases the throughput of ADB when transferring large files and also reduces latency while debugging.

Burst Mode is disabled by default. To enable the feature, do one of the following:

- Set the environment variable `ADB_BURST_MODE` to `1`.
- In Android Studio, go to the debugger settings at **File** (or **Android Studio** on macOS) **> Settings > Build, Execution, Deployment > Debugger** and set **ADB Server Burst Mode** to **Enabled**.
