Documentation

NAVIGATION

Android Rules

Rules

android_binary

```
android_binary(name, deps, srcs, aapt_version, assets, assets_dir, compatible_with
(common-definitions.html#common.compatible_with), crunch_png, custom_package, debug_key,
densities, deprecation (common-definitions.html#common.deprecation), dex_shards,
dexopts, distribs (common-definitions.html#common.distribs), enable_data_binding,
features (common-definitions.html#common.features), incremental_dexing,
inline_constants, instruments, javacopts, licenses (common-
definitions.html#common.licenses), main_dex_list, main_dex_list_opts,
main_dex_proguard_specs, manifest, manifest_merger, manifest_values, multidex,
nocompress_extensions, plugins, proguard_apply_dictionary, proguard_apply_mapping,
proguard_generate_mapping, proguard_specs, resource_configuration_filters,
resource_files, restricted_to (common-definitions.html#common.restricted_to),
shrink_resources, tags (common-definitions.html#common.tags), testonly (common-
definitions.html#common.testonly), visibility (common-
definitions.html#common.visibility))
```

Produces Android application package files (.apk).

Implicit output targets

- `name.apk` : An Android application package file signed with debug keys and zipaligned (http://developer.android.com/guide/developing/tools/zipalign.html), it could be used to develop and debug your application. You cannot release your application when signed with the debug keys.
- `name_unsigned.apk` : An unsigned version of the above file that could be signed with the release keys before release to the public.
- `name_deploy.jar` : A Java archive containing the transitive closure of this target.
  The deploy jar contains all the classes that would be found by a classloader that searched the runtime classpath of this target from beginning to end.
- `name_proguard.jar` : A Java archive containing the result of running ProGuard on the `name_deploy.jar` . This output is only produced if proguard_specs (android.html#android_binary.proguard_specs) attribute is specified.
- `name_proguard.map` : A mapping file result of running ProGuard on the `name_deploy.jar` . This output is only produced if proguard_specs (android.html#android_binary.proguard_specs) attribute is specified and

proguard_generate_mapping (android.html#android_binary.proguard_generate_mapping) or shrink_resources (android.html#android_binary.shrink_resources) is set.

Examples

Examples of Android rules can be found in the `examples/android` directory of the Bazel source tree.

Arguments

Attributes	
`name`	`Name (../build-ref.html#name); required` A unique name for this rule.
`deps`	`List of labels (../build-ref.html#labels); optional` The list of other libraries to be linked in to the binary target. Permitted library types are: `android_library`, `java_library` with `android` constraint and `cc_library` wrapping or producing `.so` native libraries for the Android target platform.
`srcs`	`List of labels (../build-ref.html#labels); optional` The list of source files that are processed to create the target. `srcs` files of type `.java` are compiled. *For readability's sake*, it is not good to put the name of a generated `.java` source file into the `srcs`. Instead, put the depended-on rule name in the `srcs`, as described below. `srcs` files of type `.srcjar` are unpacked and compiled. (This is useful if you need to generate a set of .java files with a genrule or build extension.) This rule currently forces source and class compatibility with Java 6.

Attributes	
`aapt_version`	`String; optional; default is "auto"` Select the version of aapt for this rule. Possible values: • `aapt_version = "aapt"` : Use aapt. This is the current default behaviour, and should be used for production binaries. The android_sdk rule must have an aapt binary to use this option. • `aapt_version = "aapt2"` : Use aapt2. This is the new resource packaging system that provides improved incremental resource processing, smaller apks and more. The android_sdk rule must have the aapt2 binary to use this option. • `aapt_version = "auto"` : aapt is controlled by the --android_aapt flag.
`assets`	`List of labels (../build-ref.html#labels); optional` The list of assets to be packaged. This is typically a `glob` of all files under the `assets` directory. You can also reference other rules (any rule that produces files) or exported files in the other packages, as long as all those files are under the `assets_dir` directory in the corresponding package.
`assets_dir`	`String; optional` The string giving the path to the files in `assets` . The pair `assets` and `assets_dir` describe packaged assets and either both attributes should be provided or none of them.
`crunch_png`	`Boolean; optional; default is 1` Do PNG crunching (or not). This is independent of nine-patch processing, which is always done. Currently only supported for local resources (not android_resources).

Attributes	
`custom_package`	`String; optional` Java package for which java sources will be generated. By default the package is inferred from the directory where the BUILD file containing the rule is. You can specify a different package but this is highly discouraged since it can introduce classpath conflicts with other libraries that will only be detected at runtime.
`debug_key`	`Label (../build-ref.html#labels); optional; default is @bazel_tools//tools/android:debug_keystore` File containing the debug keystore to be used to sign the debug apk. Usually you do not want to use a key other than the default key, so this attribute should be omitted. *WARNING: Do not use your production keys, they should be strictly safeguarded and not kept in your source tree.*
`densities`	`List of strings; optional` Densities to filter for when building the apk. This will strip out raster drawable resources that would not be loaded by a device with the specified screen densities, to reduce APK size. A corresponding compatible-screens section will also be added to the manifest if it does not already contain a superset listing.
`dex_shards`	`Integer; optional; default is 1` Number of shards dexing should be decomposed into. This is makes dexing much faster at the expense of app installation and startup time. The larger the binary, the more shards should be used. 25 is a good value to start experimenting with. Note that each shard will result in at least one dex in the final app. For this reason, setting this to more than 1 is not recommended for release binaries.

Attributes	
`dexopts`	`List of strings; optional` Additional command-line flags for the dx tool when generating classes.dex. Subject to "Make variable" (make-variables.html) substitution and Bourne shell tokenization (common-definitions.html#sh-tokenization).
`enable_data_binding`	`Boolean; optional; default is 0` If true, this rule processes data binding (https://developer.android.com/topic/libraries/data-binding/index.html) expressions in layout resources included through the resource_files (android.html#android_binary.resource_files) attribute. Without this setting, data binding expressions produce build failures. To build an Android app with data binding, you must also do the following: 1. Set this attribute for all Android rules that transitively depend on this one. This is because dependers inherit the rule's data binding expressions through resource merging. So they also need to build with data binding to parse those expressions. 2. Add a `deps =` entry for the data binding runtime library to all targets that set this attribute. The location of this library depends on your depot setup.
`incremental_dexing`	`Integer; optional; nonconfigurable (common-definitions.html#configurable-attributes); default is -1` Force the target to be built with or without incremental dexing, overriding defaults and --incremental_dexing flag.
`inline_constants`	`Boolean; optional; default is 0` Let the compiler inline the constants defined in the generated java sources. This attribute must be set to 0 for all `android_library` rules used directly by an `android_binary`, and for any `android_binary` that has an `android_library` in its transitive closure.

Attributes	
`instruments`	`Label (../build-ref.html#labels); optional` The `android_binary` target to instrument. If this attribute is set, this `android_binary` will be treated as a test application for instrumentation tests. An `android_instrumentation_test` target can then specify this target in its test_app (android.html#android_instrumentation_test.test_app) attribute.
`javacopts`	`List of strings; optional` Extra compiler options for this target. Subject to "Make variable" (make-variables.html) substitution and Bourne shell tokenization (common-definitions.html#sh-tokenization). These compiler options are passed to javac after the global compiler options.

`main_dex_list`	`Label (../build-ref.html#labels); optional`
	
	A text file contains a list of class file names. Classes defined by those class files are put in the primary classes.dex. e.g.:
	
	`android/support/multidex/MultiDex$V19.class`
	`android/support/multidex/MultiDex.class`
	`android/support/multidex/MultiDexApplication.class`
	`com/google/common/base/Objects.class`
	
	Must be used with `multidex="manual_main_dex"`.

`main_dex_list_opts`	`List of strings; optional` Command line options to pass to the main dex list builder. Use this option to affect the classes included in the main dex list.

Attributes	
`main_dex_proguard_specs`	`List of labels (../build-ref.html#labels); optional` Files to be used as the Proguard specifications to determine classes that must be kept in the main dex. Only allowed if the `multidex` attribute is set to `legacy`.
`manifest`	`Label (../build-ref.html#labels); optional` The name of the Android manifest file, normally `AndroidManifest.xml`. Must be defined if resource_files or assets are defined.
`manifest_merger`	`String; optional; default is "auto"` Select the manifest merger to use for this rule. Possible values: • `manifest_merger = "legacy"` : Use the legacy manifest merger. Does not allow features of the android merger like placeholder substitution and tools attributes for defining merge behavior. Removes all `<uses-permission>` and `<uses-permission-sdk-23>` tags. Performs a tag-level merge. • `manifest_merger = "android"` : Use the android manifest merger. Allows features like placeholder substitution and tools attributes for defining merge behavior. Follows the semantics from the documentation (https://developer.android.com/studio/build/manifest-merge.html) except it has been modified to also remove all `<uses-permission>` and `<uses-permission-sdk-23>` tags. Performs an attribute-level merge. • `manifest_merger = "auto"` : Merger is controlled by the --android_manifest_merger (../user-manual.html#flag--android_manifest_merger) flag.

Attributes	
`manifest_values`	`Dictionary: String -> String; optional` A dictionary of values to be overridden in the manifest. Any instance of ${name} in the manifest will be replaced with the value corresponding to name in this dictionary. applicationId, versionCode, versionName, minSdkVersion, targetSdkVersion and maxSdkVersion will also override the corresponding attributes of the manifest and uses-sdk tags. packageName will be ignored and will be set from either applicationId if specified or the package in manifest. When manifest_merger is set to legacy, only applicationId, versionCode and versionName will have any effect.
`multidex`	`String; optional; default is "off"` Whether to split code into multiple dex files. Possible values: • `native` : Split code into multiple dex files when the dex 64K index limit is exceeded. Assumes native platform support for loading multidex classes at runtime. *This works with only Android L and newer*. • `legacy` : Split code into multiple dex files when the dex 64K index limit is exceeded. Assumes multidex classes are loaded through application code (i.e. no native platform support). • `manual_main_dex` : Split code into multiple dex files when the dex 64K index limit is exceeded. The content of the main dex file needs to be specified by providing a list of classes in a text file using the main_dex_list (android.html#android_binary.main_dex_list) attribute. • `off` : Compile all code to a single dex file, even if it exceeds the index limit.
`nocompress_extensions`	`List of strings; optional` A list of file extension to leave uncompressed in apk.

Attributes	
`plugins`	`List of labels (../build-ref.html#labels); optional` Java compiler plugins to run at compile-time. Every `java_plugin` specified in the plugins attribute will be run whenever this target is built. Resources generated by the plugin will be included in the result jar of the target.
`proguard_apply_dictionary`	`Label (../build-ref.html#labels); optional` File to be used as a mapping for proguard. A line separated file of "words" to pull from when renaming classes and members during obfuscation.
`proguard_apply_mapping`	`Label (../build-ref.html#labels); optional` File to be used as a mapping for proguard. A mapping file generated by `proguard_generate_mapping` to be re-used to apply the same mapping to a new build.
`proguard_generate_mapping`	`Boolean; optional; nonconfigurable (common-definitions.html#configurable-attributes); default is 0` Whether to generate Proguard mapping file. The mapping file will be generated only if `proguard_specs` is specified. This file will list the mapping between the original and obfuscated class, method, and field names. *WARNING: If this attribute is used, the Proguard specification should contain neither `-dontobfuscate` nor `-printmapping`.*
`proguard_specs`	`List of labels (../build-ref.html#labels); optional` Files to be used as Proguard specification. This file will describe the set of specifications to be used by Proguard.
`resource_configuration_filters`	`List of strings; optional` A list of resource configuration filters, such 'en' that will limit the resources in the apk to only the ones in the 'en' configuration.

Attributes	
`resource_files`	`List of labels (../build-ref.html#labels); optional` The list of resources to be packaged. This is typically a `glob` of all files under the `res` directory. Generated files (from genrules) can be referenced by Label (../build-ref.html#labels) here as well. The only restriction is that the generated outputs must be under the same " `res` " directory as any other resource files that are included.
`shrink_resources`	`Integer; optional; default is -1` Whether to perform resource shrinking. Resources that are not used by the binary will be removed from the APK. This is only supported for rules using local resources (i.e. the `manifest` and `resource_files` attributes) and requires ProGuard. It operates in mostly the same manner as the Gradle resource shrinker (https://developer.android.com/studio/build/shrink-code.html#shrink-resources). Notable differences: • resources in `values/` will be removed as well as file based resources • uses `strict mode` by default • removing unused ID resources is not supported If resource shrinking is enabled, `*name*_files/resource_shrinker.log` will also be generated, detailing the analysis and deletions performed. Possible values: • `shrink_resources = 1` : Turns on Android resource shrinking • `shrink_resources = 0` : Turns off Android resource shrinking • `shrink_resources = -1` : Shrinking is controlled by the --android_resource_shrinking (../user-manual.html#flag--android_resource_shrinking) flag.

aar_import

```
aar_import(name, deps (common-definitions.html#common.deps), data (common-
definitions.html#common.data), aar, compatible_with (common-
definitions.html#common.compatible_with), deprecation (common-
definitions.html#common.deprecation), distribs (common-
definitions.html#common.distribs), exports, features (common-
definitions.html#common.features), licenses (common-definitions.html#common.licenses),
restricted_to (common-definitions.html#common.restricted_to), tags (common-
definitions.html#common.tags), testonly (common-definitions.html#common.testonly),
visibility (common-definitions.html#common.visibility))
```

This rule allows the use of `.aar` files as libraries for `android_library (android.html#android_library)` and `android_binary (android.html#android_binary)` rules.

Examples

```
aar_import(
    name = "google-vr-sdk",
    aar = "gvr-android-sdk/libraries/sdk-common-1.10.0.aar",
)

android_binary(
    name = "app",
    manifest = "AndroidManifest.xml",
    srcs = glob(["**.java"]),
    deps = [":google-vr-sdk"],
)
```

Arguments

Attributes

Attributes	
`name`	`Name (../build-ref.html#name); required` A unique name for this rule.
`aar`	`Label (../build-ref.html#labels); required` The `.aar` file to provide to the Android targets that depend on this target.
`exports`	`List of labels (../build-ref.html#labels); optional` Targets to export to rules that depend on this rule. See java_library.exports. (java.html#java_library.exports)

android_library

```
android_library(name, deps, srcs, data (common-definitions.html#common.data), assets,
assets_dir, compatible_with (common-definitions.html#common.compatible_with),
custom_package, deprecation (common-definitions.html#common.deprecation), distribs
(common-definitions.html#common.distribs), enable_data_binding, exported_plugins,
exports, exports_manifest, features (common-definitions.html#common.features),
idl_import_root, idl_parcelables, idl_preprocessed, idl_srcs, inline_constants,
javacopts, licenses (common-definitions.html#common.licenses), manifest, neverlink,
plugins, proguard_specs, resource_files, restricted_to (common-
definitions.html#common.restricted_to), tags (common-definitions.html#common.tags),
testonly (common-definitions.html#common.testonly), visibility (common-
definitions.html#common.visibility))
```

This rule compiles and archives its sources into a `.jar` file. The Android runtime library `android.jar` is implicitly put on the compilation class path.

Implicit output targets

- `lib`*`name`*`.jar` : A Java archive.

- `lib`*`name`*`-src.jar` : An archive containing the sources ("source jar").
- *`name`*`.aar` : An android 'aar' bundle containing the java archive and resources of this target. It does not contain the transitive closure.

Examples

Examples of Android rules can be found in the `examples/android` directory of the Bazel source tree.

The following example shows how to set `idl_import_root` . Let `//java/bazel/helloandroid/BUILD` contain:

```
android_library(
    name = "parcelable",
    srcs = ["MyParcelable.java"], # bazel.helloandroid.MyParcelable

    # MyParcelable.aidl will be used as import for other .aidl
    # files that depend on it, but will not be compiled.
    idl_parcelables = ["MyParcelable.aidl"] # bazel.helloandroid.MyParcelable

    # We don't need to specify idl_import_root since the aidl file
    # which declares bazel.helloandroid.MyParcelable
    # is present at java/bazel/helloandroid/MyParcelable.aidl
    # underneath a java root (java/).
)

android_library(
    name = "foreign_parcelable",
    srcs = ["src/android/helloandroid/OtherParcelable.java"], # android.helloandroid.OtherP
    idl_parcelables = [
        "src/android/helloandroid/OtherParcelable.aidl" # android.helloandroid.OtherParcela
    ],

    # We need to specify idl_import_root because the aidl file which
    # declares android.helloandroid.OtherParcelable is not positioned
    # at android/helloandroid/OtherParcelable.aidl under a normal java root.
    # Setting idl_import_root to "src" in //java/bazel/helloandroid
    # adds java/bazel/helloandroid/src to the list of roots
    # the aidl compiler will search for imported types.
    idl_import_root = "src",
)

# Here, OtherInterface.aidl has an "import android.helloandroid.CallbackInterface;" statem
android_library(
    name = "foreign_interface",
    idl_srcs = [
        "src/android/helloandroid/OtherInterface.aidl" # android.helloandroid.OtherInterfa
        "src/android/helloandroid/CallbackInterface.aidl" # android.helloandroid.CallbackI
    ],

    # As above, idl_srcs which are not correctly positioned under a java root
    # must have idl_import_root set. Otherwise, OtherInterface (or any other
    # interface in a library which depends on this one) will not be able
    # to find CallbackInterface when it is imported.
    idl_import_root = "src",
)

# MyParcelable.aidl is imported by MyInterface.aidl, so the generated
# MyInterface.java requires MyParcelable.class at compile time.
# Depending on :parcelable ensures that aidl compilation of MyInterface.aidl
# specifies the correct import roots and can access MyParcelable.aidl, and
# makes MyParcelable.class available to Java compilation of MyInterface.java
# as usual.
android_library(
    name = "idl",
    idl_srcs = ["MyInterface.aidl"],
    deps = [":parcelable"],
)

# Here, ServiceParcelable uses and thus depends on ParcelableService,
# when it's compiled, but ParcelableService also uses ServiceParcelable,
# which creates a circular dependency.
# As a result, these files must be compiled together, in the same android_library.
android_library(
    name = "circular_dependencies",
    srcs = ["ServiceParcelable.java"],
    idl_srcs = ["ParcelableService.aidl"],
    idl_parcelables = ["ServiceParcelable.aidl"],
)
```

Arguments

Attributes	
`name`	`Name (../build-ref.html#name); required` A unique name for this rule.
`deps`	`List of labels (../build-ref.html#labels); optional` The list of other libraries to link against. Permitted library types are: `android_library` , `java_library` with `android` constraint and `cc_library` wrapping or producing `.so` native libraries for the Android target platform.
`srcs`	`List of labels (../build-ref.html#labels); optional` The list of `.java` or `.srcjar` files that are processed to create the target. `srcs` files of type `.java` are compiled. *For readability's sake*, it is not good to put the name of a generated `.java` source file into the `srcs` . Instead, put the depended-on rule name in the `srcs` , as described below. `srcs` files of type `.srcjar` are unpacked and compiled. (This is useful if you need to generate a set of .java files with a genrule or build extension.) This rule currently forces source and class compatibility with Java 7, although try with resources is not supported. If `srcs` is omitted, then any dependency specified in `deps` is exported from this rule (see java_library's exports (java.html#java_library.exports) for more information about exporting dependencies). However, this behavior will be deprecated soon; try not to rely on it.
`assets`	`List of labels (../build-ref.html#labels); optional` The list of assets to be packaged. This is typically a `glob` of all files under the `assets` directory. You can also reference other rules (any rule that produces files) or exported files in the other packages, as long as all those files are under the `assets_dir` directory in the corresponding package.

Attributes	
`assets_dir`	`String; optional` The string giving the path to the files in `assets`. The pair `assets` and `assets_dir` describe packaged assets and either both attributes should be provided or none of them.
`custom_package`	`String; optional` Java package for which java sources will be generated. By default the package is inferred from the directory where the BUILD file containing the rule is. You can specify a different package but this is highly discouraged since it can introduce classpath conflicts with other libraries that will only be detected at runtime.
`enable_data_binding`	`Boolean; optional; default is 0` If true, this rule processes data binding (https://developer.android.com/topic/libraries/data-binding/index.html) expressions in layout resources included through the resource_files (android.html#android_binary.resource_files) attribute. Without this setting, data binding expressions produce build failures. To build an Android app with data binding, you must also do the following: 1. Set this attribute for all Android rules that transitively depend on this one. This is because dependers inherit the rule's data binding expressions through resource merging. So they also need to build with data binding to parse those expressions. 2. Add a `deps =` entry for the data binding runtime library to all targets that set this attribute. The location of this library depends on your depot setup.

Attributes	
`exported_plugins`	`List of labels (../build-ref.html#labels); optional` The list of `java_plugin` s (e.g. annotation processors) to export to libraries that directly depend on this library. The specified list of `java_plugin` s will be applied to any library which directly depends on this library, just as if that library had explicitly declared these labels in `plugins (android.html#android_library.plugins)` .
`exports`	`List of labels (../build-ref.html#labels); optional` The closure of all rules reached via `exports` attributes are considered direct dependencies of any rule that directly depends on the target with `exports` . The `exports` are not direct deps of the rule they belong to.
`exports_manifest`	`Integer; optional; default is 1` Whether to export manifest entries to `android_binary` targets that depend on this target. `uses-permissions` attributes are never exported.
`idl_import_root`	`String; optional` Package-relative path to the root of the java package tree containing idl sources included in this library. This path will be used as the import root when processing idl sources that depend on this library. When `idl_import_root` is specified, both `idl_parcelables` and `idl_srcs` must be at the path specified by the java package of the object they represent under `idl_import_root` . When `idl_import_root` is not specified, both `idl_parcelables` and `idl_srcs` must be at the path specified by their package under a Java root. See examples (android.html#android_library_examples.idl_import_root).

Attributes	
`idl_parcelables`	`List of labels (../build-ref.html#labels); optional` List of Android IDL definitions to supply as imports. These files will be made available as imports for any `android_library` target that depends on this library, directly or via its transitive closure, but will not be translated to Java or compiled. Only `.aidl` files that correspond directly to `.java` sources in this library should be included (e.g., custom implementations of Parcelable), otherwise `idl_srcs` should be used. These files must be placed appropriately for the aidl compiler to find them. See the description of idl_import_root (android.html#android_library.idl_import_root) for information about what this means.
`idl_preprocessed`	`List of labels (../build-ref.html#labels); optional` List of preprocessed Android IDL definitions to supply as imports. These files will be made available as imports for any `android_library` target that depends on this library, directly or via its transitive closure, but will not be translated to Java or compiled. Only preprocessed `.aidl` files that correspond directly to `.java` sources in this library should be included (e.g., custom implementations of Parcelable), otherwise use `idl_srcs` for Android IDL definitions that need to be translated to Java interfaces and use `idl_parcelable` for non-preprcessed AIDL files.

Attributes	
`idl_srcs`	`List of labels (../build-ref.html#labels); optional` List of Android IDL definitions to translate to Java interfaces. After the Java interfaces are generated, they will be compiled together with the contents of `srcs` . These files will be made available as imports for any `android_library` target that depends on this library, directly or via its transitive closure. These files must be placed appropriately for the aidl compiler to find them. See the description of idl_import_root (android.html#android_library.idl_import_root) for information about what this means.
`inline_constants`	`Boolean; optional; default is 0` Let the compiler inline the constants defined in the generated java sources. This attribute must be set to 0 for all `android_library` rules used directly by an `android_binary` , and for any `android_binary` that has an `android_library` in its transitive closure.
`javacopts`	`List of strings; optional` Extra compiler options for this target. Subject to "Make variable" (make-variables.html) substitution and Bourne shell tokenization (common-definitions.html#sh-tokenization). These compiler options are passed to javac after the global compiler options.
`manifest`	`Label (../build-ref.html#labels); optional` The name of the Android manifest file, normally `AndroidManifest.xml` . Must be defined if resource_files or assets are defined.
`neverlink`	`Boolean; optional; default is 0` Only use this library for compilation and not at runtime. The outputs of a rule marked as `neverlink` will not be used in `.apk` creation. Useful if the library will be provided by the runtime environment during execution.

Attributes	
`plugins`	`List of labels (../build-ref.html#labels); optional` Java compiler plugins to run at compile-time. Every `java_plugin` specified in the plugins attribute will be run whenever this target is built. Resources generated by the plugin will be included in the result jar of the target.
`proguard_specs`	`List of labels (../build-ref.html#labels); optional` Files to be used as Proguard specification. These will describe the set of specifications to be used by Proguard. If specified, they will be added to any `android_binary` target depending on this library. The files included here must only have idempotent rules, namely -dontnote, -dontwarn, assumenosideeffects, and rules that start with -keep. Other options can only appear in `android_binary` 's proguard_specs, to ensure non-tautological merges.
`resource_files`	`List of labels (../build-ref.html#labels); optional` The list of resources to be packaged. This is typically a `glob` of all files under the `res` directory. Generated files (from genrules) can be referenced by Label (../build-ref.html#labels) here as well. The only restriction is that the generated outputs must be under the same " `res` " directory as any other resource files that are included.

android_instrumentation_test

```
android_instrumentation_test(name, data (common-definitions.html#common.data), args
(common-definitions.html#test.args), compatible_with (common-
definitions.html#common.compatible_with), deprecation (common-
definitions.html#common.deprecation), distribs (common-
definitions.html#common.distribs), features (common-definitions.html#common.features),
flaky (common-definitions.html#test.flaky), licenses (common-
definitions.html#common.licenses), local (common-definitions.html#test.local),
restricted_to (common-definitions.html#common.restricted_to), shard_count (common-
definitions.html#test.shard_count), size (common-definitions.html#test.size),
support_apks, tags (common-definitions.html#common.tags), target_device, test_app,
testonly (common-definitions.html#common.testonly), timeout (common-
definitions.html#test.timeout), toolchains, visibility (common-
definitions.html#common.visibility))
```

An `android_instrumentation_test` rule runs Android instrumentation tests. It will start an emulator, install the application being tested, the test application, and any other needed applications, and run the tests defined in the test package.

The test_app (android.html#android_instrumentation_test.test_app) attribute specifies the `android_binary` which contains the test. This `android_binary` in turn specifies the `android_binary` application under test through its instruments (${link android_binary.instruments) attribute.

Example

```
# java/com/samples/hello_world/BUILD

android_library(
    name = "hello_world_lib",
    srcs = ["Lib.java"],
    manifest = "LibraryManifest.xml",
    resource_files = glob(["res/**"]),
)

# The app under test
android_binary(
    name = "hello_world_app",
    manifest = "AndroidManifest.xml",
    deps = [":hello_world_lib"],
)
```

```
# javatests/com/samples/hello_world/BUILD

android_library(
    name = "hello_world_test_lib",
    srcs = ["Tests.java"],
    deps = [
      "//java/com/samples/hello_world:hello_world_lib",
      ...  # test dependencies such as Espresso and Mockito
    ],
)

# The test app
android_binary(
    name = "hello_world_test_app",
    instruments = "//java/com/samples/hello_world:hello_world_app",
    manifest = "AndroidManifest.xml",
    deps = ["hello_world_test_lib"],
)

android_instrumentation_test(
    name = "hello_world_uiinstrumentation_tests",
    target_device = ":some_target_device",
    test_app = ":hello_world_test_app",
)
```

Arguments

Attributes	
`name`	`Name (../build-ref.html#name); required` A unique name for this rule.

Attributes	
`support_apks`	`List of labels (../build-ref.html#labels); optional` Other APKs to install on the device before the instrumentation test starts.
`target_device`	`Label (../build-ref.html#labels); required` The android_device (android.html#android_device) the test should run on. To run the test on an emulator that is already running or on a physical device, use these arguments: `--test_output=streamed --test_arg=--device_broker_type=LOCAL_ADB_SERVER --test_arg=--device_serial_number=$device_identifier`
`test_app`	`Label (../build-ref.html#labels); required` The android_binary (android.html#android_binary) target containing the test classes. The `android_binary` target must specify which target it is testing through its `instruments` (android.html#android_binary.instruments) attribute.
`toolchains`	`List of labels (../build-ref.html#labels); optional` The set of toolchains that supply "Make variables" (make-variables.html) that this target can use in some of its attributes. Some rules have toolchains whose Make variables they can use by default.

android_local_test

```
android_local_test(name, deps, srcs, data (common-definitions.html#common.data), args
(common-definitions.html#test.args), compatible_with (common-
definitions.html#common.compatible_with), custom_package, deprecation (common-
definitions.html#common.deprecation), features (common-
definitions.html#common.features), flaky (common-definitions.html#test.flaky),
javacopts, jvm_flags, licenses (common-definitions.html#common.licenses), local (common-
definitions.html#test.local), manifest, manifest_values, plugins, resource_jars,
resource_strip_prefix, restricted_to (common-definitions.html#common.restricted_to),
runtime_deps, shard_count (common-definitions.html#test.shard_count), size (common-
definitions.html#test.size), stamp, tags (common-definitions.html#common.tags),
test_class, testonly (common-definitions.html#common.testonly), timeout (common-
definitions.html#test.timeout), toolchains, visibility (common-
definitions.html#common.visibility))
```

This rule is for unit testing `android_library` rules locally (as opposed to on a device). It works with the Android Robolectric testing framework. See the Android Robolectric (http://robolectric.org/) site for details about writing Robolectric tests.

Implicit output targets

- *name*`.jar` : A Java archive of the test.
- *name*`-src.jar` : An archive containing the sources ("source jar").

Examples

To use Robolectric with `android_local_test`, add Robolectric's repository (https://github.com/robolectric/robolectric/tree/master/bazel) to your `WORKSPACE` file:

```
http_archive(
 name = "robolectric",
 urls = ["https://github.com/robolectric/robolectric/archive/<COMMIT>.tar.gz"],
 strip_prefix = "robolectric-<COMMIT>",
 sha256 = "<HASH>",
)
load("@robolectric//bazel:robolectric.bzl", "robolectric_repositories")
robolectric_repositories()
```

This pulls in the `maven_jar` rules needed for Robolectric. Then each `android_local_test` rule should depend on `@robolectric//bazel:robolectric` . See example below.

```
android_local_test(
    name = "SampleTest",
    srcs = [
        "SampleTest.java",
    ],
    manifest = "LibManifest.xml",
    deps = [
        ":sample_test_lib",
        "@robolectric//bazel:robolectric",
    ],
)

android_library(
    name = "sample_test_lib",
    srcs = [
         "Lib.java",
    ]
    resource_files = glob(["res/**"]),
    manifest = "AndroidManifest.xml",
)
```

Arguments

Attributes	
`name`	`Name (../build-ref.html#name); required` A unique name for this rule.
`deps`	`List of labels (../build-ref.html#labels); optional` The list of libraries to be tested as well as additional libraries to be linked in to the target. All resources, assets and manifest files declared in Android rules in the transitive closure of this attribute are made available in the test. The list of allowed rules in `deps` are `android_library`, `aar_import`, `java_import`, `java_library`, and `java_lite_proto_library`.
`srcs`	`List of labels (../build-ref.html#labels); optional` The list of source files that are processed to create the target. Required except in special case described below. `srcs` files of type `.java` are compiled. *For readability's sake*, it is not good to put the name of a generated `.java` source file into the `srcs`. Instead, put the depended-on rule name in the `srcs`, as described below. `srcs` files of type `.srcjar` are unpacked and compiled. (This is useful if you need to generate a set of .java files with a genrule or build extension.) All other files are ignored, as long as there is at least one file of a file type described above. Otherwise an error is raised. The `srcs` attribute is required and cannot be empty, unless `runtime_deps` is specified.
`custom_package`	`String; optional` Java package in which the R class will be generated. By default the package is inferred from the directory where the BUILD file containing the rule is. If you use this attribute, you will likely need to use `test_class` as well.

Attributes	
`javacopts`	`List of strings; optional` Extra compiler options for this library. Subject to "Make variable" (make-variables.html) substitution and Bourne shell tokenization (common-definitions.html#sh-tokenization). These compiler options are passed to javac after the global compiler options.
`jvm_flags`	`List of strings; optional` A list of flags to embed in the wrapper script generated for running this binary. Subject to $(location) and "Make variable" (make-variables.html) substitution, and Bourne shell tokenization (common-definitions.html#sh-tokenization). The wrapper script for a Java binary includes a CLASSPATH definition (to find all the dependent jars) and invokes the right Java interpreter. The command line generated by the wrapper script includes the name of the main class followed by a `"$@"` so you can pass along other arguments after the classname. However, arguments intended for parsing by the JVM must be specified *before* the classname on the command line. The contents of `jvm_flags` are added to the wrapper script before the classname is listed. Note that this attribute has *no effect* on `*_deploy.jar` outputs.
`manifest`	`Label (../build-ref.html#labels); optional` The name of the Android manifest file, normally `AndroidManifest.xml`. Must be defined if resource_files or assets are defined or if any of the manifests from the libraries under test have a `minSdkVersion` tag in them.

Attributes	
`manifest_values`	`Dictionary: String -> String; optional` A dictionary of values to be overridden in the manifest. Any instance of ${name} in the manifest will be replaced with the value corresponding to name in this dictionary. `applicationId` , `versionCode` , `versionName` , `minSdkVersion` , `targetSdkVersion` and `maxSdkVersion` will also override the corresponding attributes of the manifest and uses-sdk tags. `packageName` will be ignored and will be set from either `applicationId` if specified or the package in the manifest. It is not necessary to have a manifest on the rule in order to use manifest_values.
`plugins`	`List of labels (../build-ref.html#labels); optional` Java compiler plugins to run at compile-time. Every `java_plugin` specified in this attribute will be run whenever this rule is built. A library may also inherit plugins from dependencies that use `exported_plugins` . Resources generated by the plugin will be included in the resulting jar of this rule.
`resource_jars`	`List of labels (../build-ref.html#labels); optional` Set of archives containing Java resources. If specified, the contents of these jars are merged into the output jar.
`resource_strip_prefix`	`String; optional` The path prefix to strip from Java resources. If specified, this path prefix is stripped from every file in the `resources` attribute. It is an error for a resource file not to be under this directory. If not specified (the default), the path of resource file is determined according to the same logic as the Java package of source files. For example, a source file at `stuff/java/foo/bar/a.txt` will be located at `foo/bar/a.txt` .

Attributes	
`runtime_deps`	`List of labels (../build-ref.html#labels); optional` Libraries to make available to the final binary or test at runtime only. Like ordinary `deps`, these will appear on the runtime classpath, but unlike them, not on the compile-time classpath. Dependencies needed only at runtime should be listed here. Dependency-analysis tools should ignore targets that appear in both `runtime_deps` and `deps`.
`stamp`	`Integer; optional; default is 0` Enable link stamping. Whether to encode build information into the binary. Possible values: • `stamp = 1`: Stamp the build information into the binary. Stamped binaries are only rebuilt when their dependencies change. Use this if there are tests that depend on the build information. • `stamp = 0`: Always replace build information by constant values. This gives good build result caching. • `stamp = -1`: Embedding of build information is controlled by the --[no]stamp (../user-manual.html#flag--stamp) flag.
`test_class`	`String; optional` The Java class to be loaded by the test runner. This attribute specifies the name of a Java class to be run by this test. It is rare to need to set this. If this argument is omitted, the Java class whose name corresponds to the `name` of this `android_local_test` rule will be used. The test class needs to be annotated with `org.junit.runner.RunWith`.
`toolchains`	`List of labels (../build-ref.html#labels); optional` The set of toolchains that supply "Make variables" (make-variables.html) that this target can use in some of its attributes. Some rules have toolchains whose Make variables they can use by default.

android_device

```
android_device(name, cache, compatible_with (common-
definitions.html#common.compatible_with), default_properties, deprecation (common-
definitions.html#common.deprecation), distribs (common-
definitions.html#common.distribs), features (common-definitions.html#common.features),
horizontal_resolution, licenses (common-definitions.html#common.licenses),
platform_apks, ram, restricted_to (common-definitions.html#common.restricted_to),
screen_density, system_image, tags (common-definitions.html#common.tags), testonly
(common-definitions.html#common.testonly), vertical_resolution, visibility (common-
definitions.html#common.visibility), vm_heap)
```

This rule creates an android emulator configured with the given specifications. This emulator may be started via a bazel run command or by executing the generated script directly. It is encouraged to depend on existing android_device rules rather than defining your own.

This rule is a suitable target for the --run_under flag to bazel test and bazel run. It starts an emulator, copies the target being tested/run to the emulator, and tests it or runs it as appropriate.

`android_device` supports creating KVM images if the underlying system_image (android.html#android_device.system_image) is X86 based and is optimized for at most the I686 CPU architecture. To use KVM add `tags = ['requires-kvm']` to the `android_device` rule.

Implicit output targets

- *name*`_images/userdata.dat` : Contains image files and snapshots to start the emulator
- *name*`_images/emulator-meta-data.pb` : Contains serialized information necessary to pass on to the emulator to restart it.

Examples

The following example shows how to use android_device. `//java/android/helloandroid/BUILD` contains

```
android_device(
    name = "nexus_s",
    cache = 32,
    default_properties = "nexus_s.properties",
    horizontal_resolution = 480,
    ram = 512,
    screen_density = 233,
    system_image = ":emulator_images_android_16_x86",
    vertical_resolution = 800,
    vm_heap = 32,
)

filegroup(
    name = "emulator_images_android_16_x86",
    srcs = glob(["androidsdk/system-images/android-16/**"]),
)
```

`//java/android/helloandroid/nexus_s.properties` contains:

```
ro.product.brand=google
ro.product.device=crespo
ro.product.manufacturer=samsung
ro.product.model=Nexus S
ro.product.name=soju
```

This rule will generate images and a start script. You can start the emulator locally by executing bazel run :nexus_s -- --action=start. The script exposes the following flags:

- --adb_port: The port to expose adb on. If you wish to issue adb commands to the emulator this is the port you will issue adb connect to.
- --emulator_port: The port to expose the emulator's telnet management console on.
- --enable_display: Starts the emulator with a display if true (defaults to false).
- --action: Either start or kill.
- --apks_to_install: a list of apks to install on the emulator.

Arguments

Attributes	
`name`	`Name (../build-ref.html#name); required` A unique name for this rule.
`cache`	`Integer; required; default is 0` The size in megabytes of the emulator's cache partition. The minimum value of this is 16 megabytes.
`default_properties`	`Label (../build-ref.html#labels); optional` A single properties file to be placed in /default.prop on the emulator. This allows the rule author to further configure the emulator to appear more like a real device (In particular controlling its UserAgent strings and other behaviour that might cause an application or server to behave differently to a specific device). The properties in this file will override read only properties typically set by the emulator such as ro.product.model.
`horizontal_resolution`	`Integer; required; default is 0` The horizontal screen resolution in pixels to emulate. The minimum value is 240.
`platform_apks`	`List of labels (../build-ref.html#labels); optional` A list of apks to be installed on the device at boot time.
`ram`	`Integer; required; default is 0` The amount of ram in megabytes to emulate for the device. This is for the entire device, not just for a particular app installed on the device. The minimum value is 64 megabytes.
`screen_density`	`Integer; required; default is 0` The density of the emulated screen in pixels per inch. The minimum value of this is 30 ppi.

Attributes	
`system_image`	`Label (../build-ref.html#labels); required` A filegroup containing the following files: • system.img: The system partition • kernel-qemu: The Linux kernel the emulator will load • ramdisk.img: The initrd image to use at boot time • userdata.img: The initial userdata partition • source.properties: A properties file containing information about the images These files are part of the android sdk or provided by third parties (for example Intel provides x86 images).
`vertical_resolution`	`Integer; required; default is 0` The vertical screen resolution in pixels to emulate. The minimum value is 240.
`vm_heap`	`Integer; required; default is 0` The size in megabytes of the virtual machine heap Android will use for each process. The minimum value is 16 megabytes.

android_ndk_repository

```
android_ndk_repository(name, api_level, path)
```

Configures Bazel to use an Android NDK to support building Android targets with native code. NDK versions 10 up to 16 are currently supported.

Note that building for Android also requires an `android_sdk_repository` rule in your `WORKSPACE` file.

Examples

```
android_ndk_repository(
    name = "androidndk",
)
```

The above example will locate your Android NDK from `$ANDROID_NDK_HOME` and detect the highest API level that it supports.

```
android_ndk_repository(
    name = "androidndk",
    path = "./android-ndk-r12b",
    api_level = 24,
)
```

The above example will use the Android NDK located inside your workspace in `./android-ndk-r12b`. It will use the API level 24 libraries when compiling your JNI code.

cpufeatures

The Android NDK contains the cpufeatures library (https://developer.android.com/ndk/guides/cpu-features.html) which can be used to detect a device's CPU at runtime. The following example demonstrates how to use cpufeatures with Bazel.

```
# jni.cc
#include "ndk/sources/android/cpufeatures/cpu-features.h"
...

# BUILD
cc_library(
    name = "jni",
    srcs = ["jni.cc"],
    deps = ["@androidndk//:cpufeatures"],
)
```

Arguments

Attributes	
`name`	`Name (../build-ref.html#name); required` A unique name for this rule.
`api_level`	`Integer; optional; nonconfigurable (common-definitions.html#configurable-attributes); default is 0` The Android API level to build against. If not specified, the highest API level installed will be used.
`path`	`String; optional; nonconfigurable (common-definitions.html#configurable-attributes)` An absolute or relative path to an Android NDK. Either this attribute or the `$ANDROID_NDK_HOME` environment variable must be set. The Android NDK can be downloaded from the Android developer site (https://developer.android.com/ndk/downloads/index.html).

android_sdk_repository

```
android_sdk_repository(name, api_level, build_tools_version, path)
```

Configures Bazel to use a local Android SDK to support building Android targets.

Examples

The minimum to set up an Android SDK for Bazel is to put an `android_sdk_repository` rule named "androidsdk" in your `WORKSPACE` file and set the `$ANDROID_HOME` environment variable to the path of your Android SDK. Bazel will use the highest Android API level and build tools version installed in the Android SDK by default.

```
android_sdk_repository(
    name = "androidsdk",
)
```

To ensure reproducible builds, the `path`, `api_level` and `build_tools_version` attributes can be set to specific values. The build will fail if the Android SDK does not have the specified API level or build tools version installed.

```
android_sdk_repository(
    name = "androidsdk",
    path = "./sdk",
    api_level = 19,
    build_tools_version = "25.0.0",
)
```

The above example also demonstrates using a workspace-relative path to the Android SDK. This is useful if the Android SDK is part of your Bazel workspace (e.g. if it is checked into version control).

Support Libraries

The Support Libraries are available in the Android SDK Manager as "Android Support Repository". This is a versioned set of common Android libraries, such as the Support and AppCompat libraries, that is packaged as a local Maven repository. `android_sdk_repository` generates Bazel targets for each of these libraries that can be used in the dependencies of `android_binary` and `android_library` targets.

The names of the generated targets are derived from the Maven coordinates of the libraries in the Android Support Repository, formatted as `@androidsdk//${group}:${artifact}-${version}`. The following example shows how an `android_library` can depend on version 25.0.0 of the v7 appcompat library.

```
android_library(
    name = "lib",
    srcs = glob(["*.java"]),
    manifest = "AndroidManifest.xml",
    resource_files = glob(["res/**"]),
    deps = ["@androidsdk//com.android.support:appcompat-v7-25.0.0"],
)
```

Arguments

Attributes	
`name`	`Name (../build-ref.html#name); required` A unique name for this rule.
`api_level`	`Integer; optional; nonconfigurable (common-definitions.html#configurable-attributes); default is 0` The Android API level to build against by default. If not specified, the highest API level installed will be used. The API level used for a given build can be overridden by the `android_sdk` flag. `android_sdk_repository` creates an `android_sdk` target for each API level installed in the SDK with name `@androidsdk//:sdk-${level}` , whether or not this attribute is specified. For example, to build against a non-default API level: `bazel build --android_sdk=@androidsdk//:sdk-19 //java/com/example:app` . To view all `android_sdk` targets generated by `android_sdk_repository` , you can run `bazel query "kind(android_sdk, @androidsdk//...)"` .

Attributes	
`build_tools_version`	`String; optional; nonconfigurable (common-definitions.html#configurable-attributes)` The version of the Android build tools to use from within the Android SDK. If not specified, the latest build tools version installed will be used. Bazel requires build tools version 26.0.1 or later.
`path`	`String; optional; nonconfigurable (common-definitions.html#configurable-attributes)` An absolute or relative path to an Android SDK. Either this attribute or the `$ANDROID_HOME` environment variable must be set. The Android SDK can be downloaded from the Android developer site (https://developer.android.com).

Twitter (https://twitter.com/bazelbuild)
Blog (https://blog.bazel.build)
GitHub (https://github.com/bazelbuild/bazel)
Discussion group (https://groups.google.com/forum/#!forum/bazel-discuss)