Bazel Blog

How Android Builds Work in Bazel

14 February 2018

Background: How Bazel Works

In Bazel, `BUILD` files in directories specify **targets** that can be built from the contents of those directories.

Bazel goes through three steps (https://docs.bazel.build/versions/master/user-manual.html#phases) when building targets:

1. In the **loading** phase, Bazel parses the `BUILD` file of the target being built and all `BUILD` files that file transitively depends on.
2. In the **analysis** phase, Bazel builds a graph of actions needed to build the specified targets.
3. In the **execution** phase, Bazel runs those actions.

Each Bazel target is defined by a **rule**, which specifies inputs, outputs, and how to get from one to the other. Rules can specify things like creating an executable binary or defining a library. In Bazel's code, individual rules are represented by instances of implementations of `RuleConfiguredTargetFactory` (https://github.com/bazelbuild/bazel/blob/master/src/main/java/com/google/devtools/build/lib/analysis/RuleConfiguredTargetFactory.java). Users can also extend Bazel and create new rules with Skylark (https://docs.bazel.build/versions/master/skylark/concepts.html).

Rules create, in turn, any number of **action**s. Each action takes any number of **artifact**s as inputs and produces one or more artifacts as outputs. These artifacts represent files that may not yet be available. They can either be *source artifacts*, such as source code checked in to the repository, or *generated artifacts*, such as output of other actions. For example, an action to compile a piece of code might take in source artifacts representing the code to be compiled and generated artifacts representing compiled dependencies, even though those dependencies have not yet been compiled, and output a generated artifact representing the compiled result. Additionally, rules may expose any number of **provider** objects. These providers are the API rules provide to other rules. They provide read-only information about internal state.

During analysis, Bazel runs the rules for each target being built and their transitive dependencies. Each rule generates and records all the actions it depends on. Bazel won't necessarily run all of those actions; if an action doesn't end up being required, Bazel will just ignore it. **Skyframe** is used to evaluate and cache the results of rules.

Information from the rules, including artifacts representing the future output of actions, are made available to other rules through the rules' providers. Each rule has access to its direct dependencies' providers. Because most information passed between rules is actually transitive, providers make use of the `NestedSet` (https://github.com/bazelbuild/bazel/blob/master/src/main/java/com/google/devtools/build/lib/collect/nestedset/NestedSet.java) class, a DAG-like data structure (it's not actually a set!) made up of items and pointers to other (nested) `NestedSet` objects. `NestedSet` s are specially optimized to work efficiently for analysis. For part of a provider that represents some transitive state, for example, a trivial implementation might be to build a new list that contains the items for the current rule and each transitive dependency (for a chain of n transitive dependencies, that means we'd add n + (n - 1) + … 1 = O(n^2) items to some list), but building a nested set containing the new item and a pointer to the previous nested set is much more efficient (we'd add 2 + 2 + … 2 = O(n) items to some nested set). This introduces similar efficiency in memory usage as well.

Artifacts can each be added to any number of **output groups**. Each output group represents a different group of outputs that a user might choose to build. For example, the `source_jars` (https://github.com/bazelbuild/bazel/blob/master/src/main/java/com/google/devtools/build/lib/rules/java/JavaSemantics.java#L132) output group specifies that Bazel should also produce the JARs of the source for a Java target and its transitive dependencies. The special default (https://github.com/bazelbuild/bazel/blob/master/src/main/java/com/google/devtools/build/lib/analysis/OutputGroupInfo.java#L113) output group holds output that is specifically built for a target - for example, building a Java binary might produce a compiled `.jar` file in the default output group.

During execution, Bazel first looks at the artifacts in the requested output groups (plus, unless the user explicitly requested otherwise, the default output group). For each of those artifacts, it finds the actions that generate the artifact, then each of the artifacts each of those actions need, and so on until it finds all the actions and artifacts needed. If the action is not cached or the cache entry should be invalidated, Bazel follows this same process for the action's dependencies, then runs the action. Once all of the actions in the requested output groups has been run or returned from cache, the build is complete.

Android Builds

There are a few important kinds of rules when building code for Android (https://docs.bazel.build/versions/master/be/android.html):

- `android_binary` (https://docs.bazel.build/versions/master/be/android.html#android_binary) rules build Android packages (`.apk` files)
- `android_library` (https://docs.bazel.build/versions/master/be/android.html#android_library) rules build individual libraries that binaries and other libraries can consume.
- `android_local_test` (https://docs.bazel.build/versions/master/be/android.html#android_local_test) rules run test on Android code in a JVM.
- `aar_import` (https://docs.bazel.build/versions/master/be/android.html#aar_import) rules import `.aar` libraries built outside of Bazel into a Bazel target.

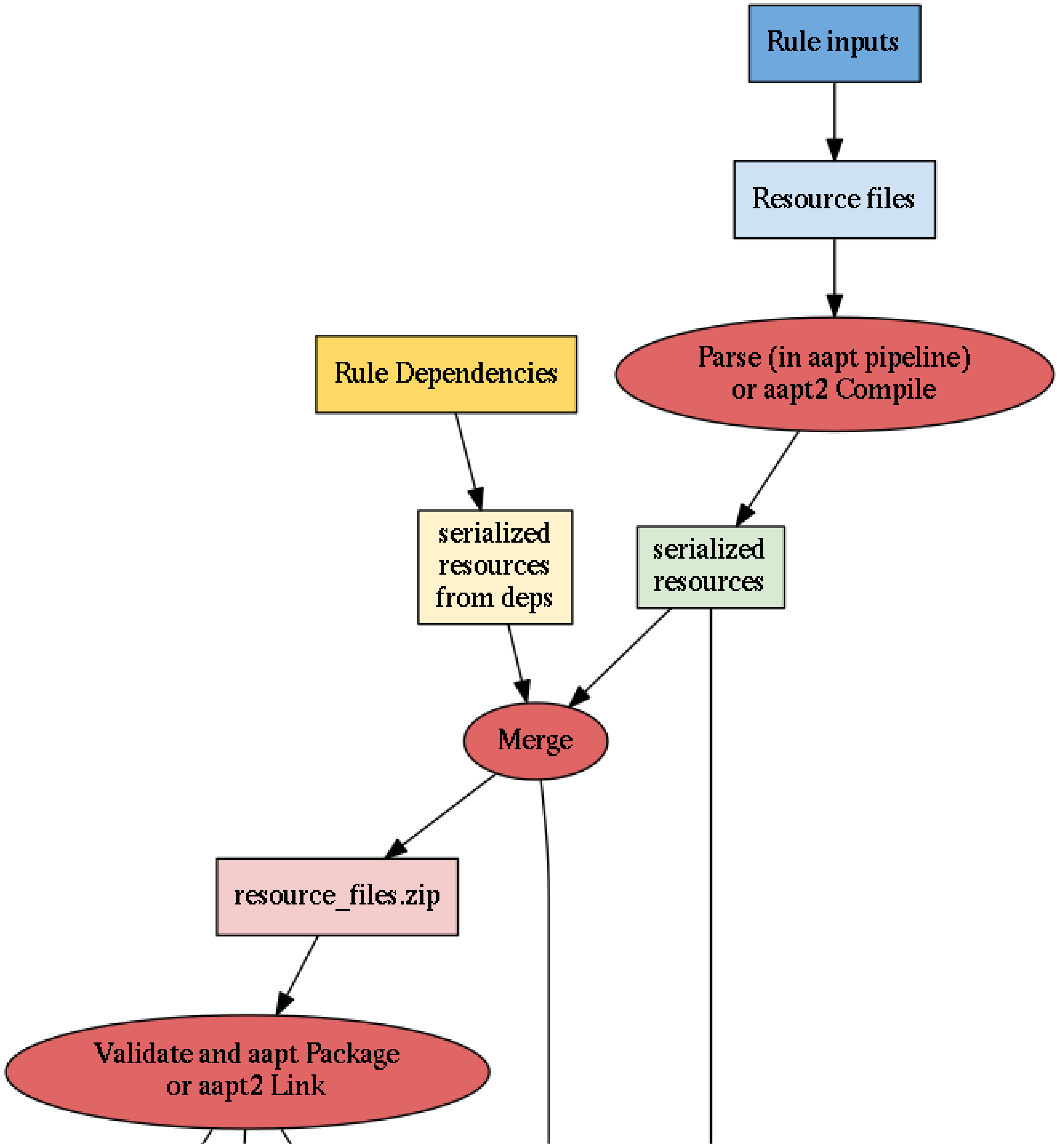
Android Resources

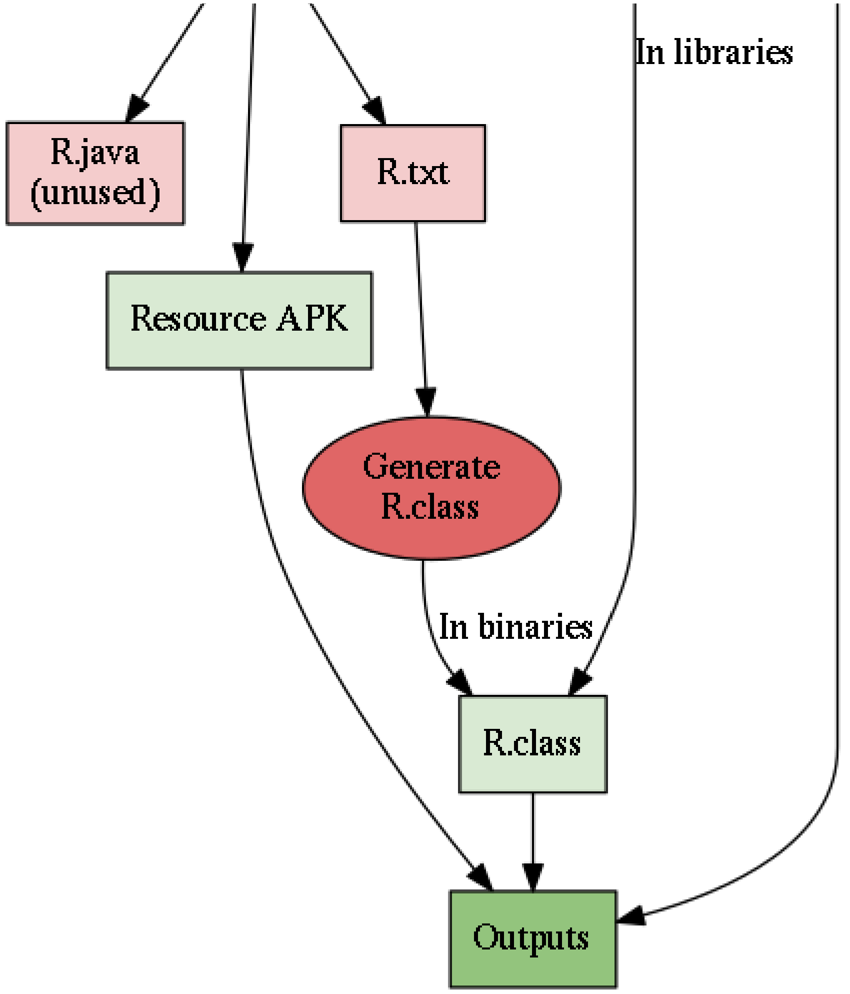
One way in which the Android library build process differs from the normal Java build process is Android **resources** (https://developer.android.com/guide/topics/resources/index.html). Resources are anything that's not code - strings, images, layouts, and so on (https://developer.android.com/guide/topics/resources/providing-resources.html#ResourceTypes).

Bazel generates **`R.java`** files (as well as related `R.class` and `R.txt` files) to contain references to available resources. These R files contain integer **resource ID**s that developers can use to refer to their resources. Within an app, each resource ID refers to one unique resource.

Developers can provide different versions (https://developer.android.com/guide/topics/resources/providing-resources.html#AlternativeResources) of the same resource (to support, for example, different languages, regions, or screen sizes). Android makes references to the base resource available in the R files, and Android devices select the best available version of that resource at runtime.

Resource processing with `aapt` and `aapt2`


Rule inputs
Resource files
Parse (in aapt pipeline) or aapt2 Compile
Rule Dependencies
serialized resources from deps
serialized resources
Merge
resource_files.zip
Validate and aapt Package or aapt2 Link



Bazel supports processing resources using the original Android resource processor, `aapt`, or the new version, `aapt2`. Both methods are fundamentally similar but have a few important differences.

Bazel goes through three steps to build resources.

First, Bazel serializes the files that define the resources. In the `aapt` pipeline, the **parse** action serializes information about resources into `symbols.bin` files. In the `aapt2` pipeline, an action calls into the **`aapt2` compile** command which serializes the information into a format used by `aapt2`.

Next, the serialized resources are **merged** with similarly serialized resources inherited from dependencies. Conflicts between identically named resources are identified and, if possible, resolved during this merging. The contents of `values` resource files are generally explicitly merged. For other files, if resources from the target or its dependencies have the same name and qualifiers, the contents of the files are compared and, if they are different, a warning is produced and the resource that was provided last is chosen to be used.

Finally, Bazel checks that the resources for the target are reasonable and packages them up. In the `aapt` pipeline, the **validate** action calls into the `aapt` **package** command, and in the `aapt2` pipeline, the **`aapt2` link** command is called. In both cases, any malformed resources or references to unavailable resources cause a failure, and, if no failures are encountered, `R.java` and `R.txt` files are produced with information about the validated resources, and a Resource APK containing those resources is produced.

Using `aapt2` rather than `aapt` provides better and more efficient support for a variety of cases. Additionally, more of the resource processing steps are handled by `aapt2` as opposed to Bazel's custom resource processing tools. Finally, since the serialized format can be understood as-is by future calls to `aapt2` , Bazel no longer has to deserialize information about resources to a form `aapt2` can understand.

The resource ID values generated for `android_library` targets are only temporary, since higher-level targets might depend on multiple targets where different resources were assigned the same ID. To ensure that resource IDs aren't persisted anywhere permanent, the R files record the IDs as nonfinal, ensuring that compilation doesn't inline them into other Java code. Additionally, an `android_library` 's R files should be discarded after building is complete.

(Even though `android_library` files are eventually discarded, we still need to run resource processing to generate a temporary `R.class` to allow compilation, to merge resources so they can be inherited by consumers, and to validate that the resources can be compiled correctly - otherwise, if a developer introduces a bug in their resource definitions, it won't be caught until they're used in an `android_binary,` resulting in a lot of wasted work done by Bazel.)

Code in android libraries and binaries make references to code in the R files, so the `R.class` file must be generated before regular compilation can start. For `android_library` targets, since all resource IDs are temporary anyway, we can speed things up by generating a `R.class` file at the end of resource merging. For `android_binary` targets, we need to wait for the output of validation to get correct resource IDs. Validation does produce an `R.java` file, but **generating an `R.class`** file directly from the contents of the `R.txt` file is much faster than compiling the `R.java` file into an `R.class` file.

Android Resource Optimizations

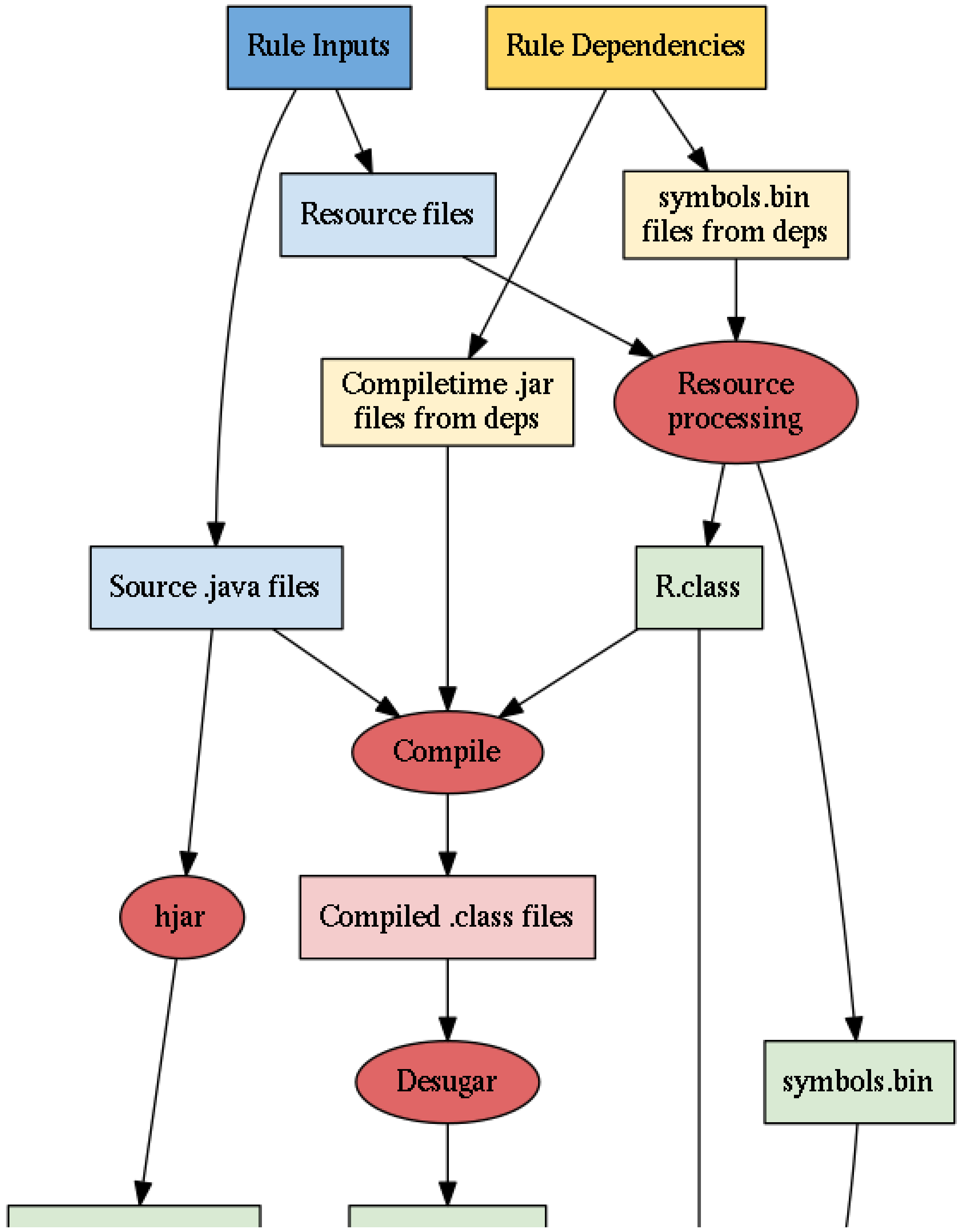
Resource Filtering

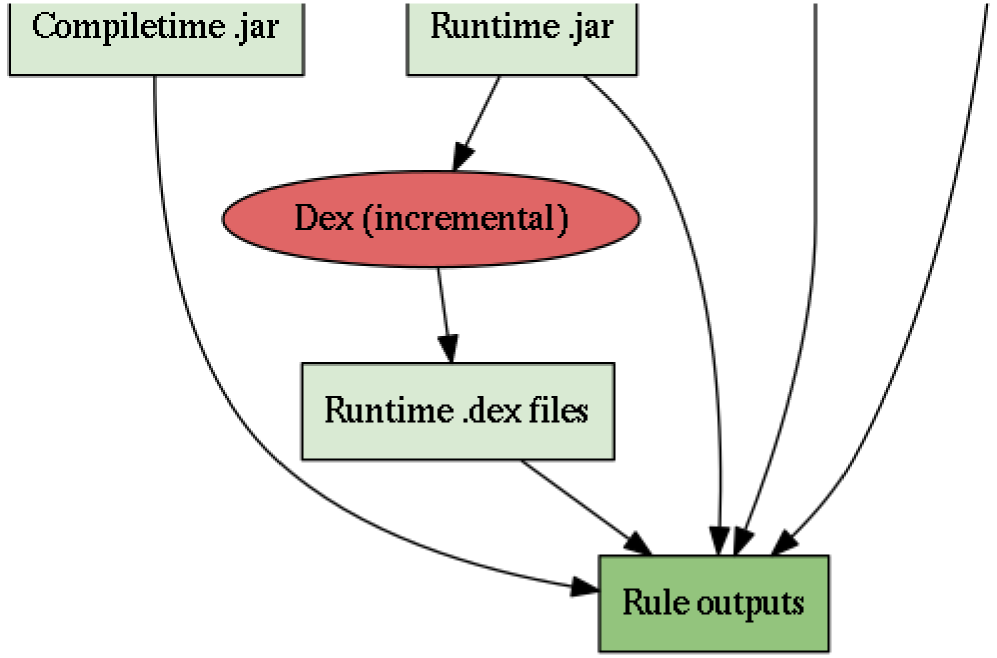
The `android_binary` rule includes optional `resource_configuration_filters` (https://docs.bazel.build/versions/master/be/android.html#android_binary.resource_configuration_filters) and `densities` (https://docs.bazel.build/versions/master/be/android.html#android_binary.densities) fields. These fields limit the types of devices that will be built for. For example, if you only wanted to build for English-language devices with HDPI displays, you could specify:

```
android_binary(
  # ...
  densities = ["hdpi"],
  resource_configuration_filters = ["en"],
)
```

Bazel will now be able to skip unneeded resources. As a result, the build will be faster and the resulting APK will be smaller. It won't support all kinds of devices and user preferences, but this speed improvement means developers can build and iterate faster.

Android Libraries


Rule Inputs
Rule Dependencies
Resource files
symbols.bin files from deps
Compiletime .jar files from deps
Resource processing
Source .java files
R.class
Compile
hjar
Compiled .class files
Desugar
symbols.bin



An `android_library` (https://docs.bazel.build/versions/master/be/android.html#android_library) rule is a pretty simple rule that builds and organizes an android library for use in another Android target. In the analysis phase, there are basically three groups of actions generated:

First, Bazel processes the library's resources, as described above.

Next comes the actual compilation of the library. This mostly just uses the regular Bazel Java compilation path. The biggest difference is that the `R.class` file produced in resource processing is also included in the compilation path (but is not inherited by consumers, since the R files need to be regenerated for each target).

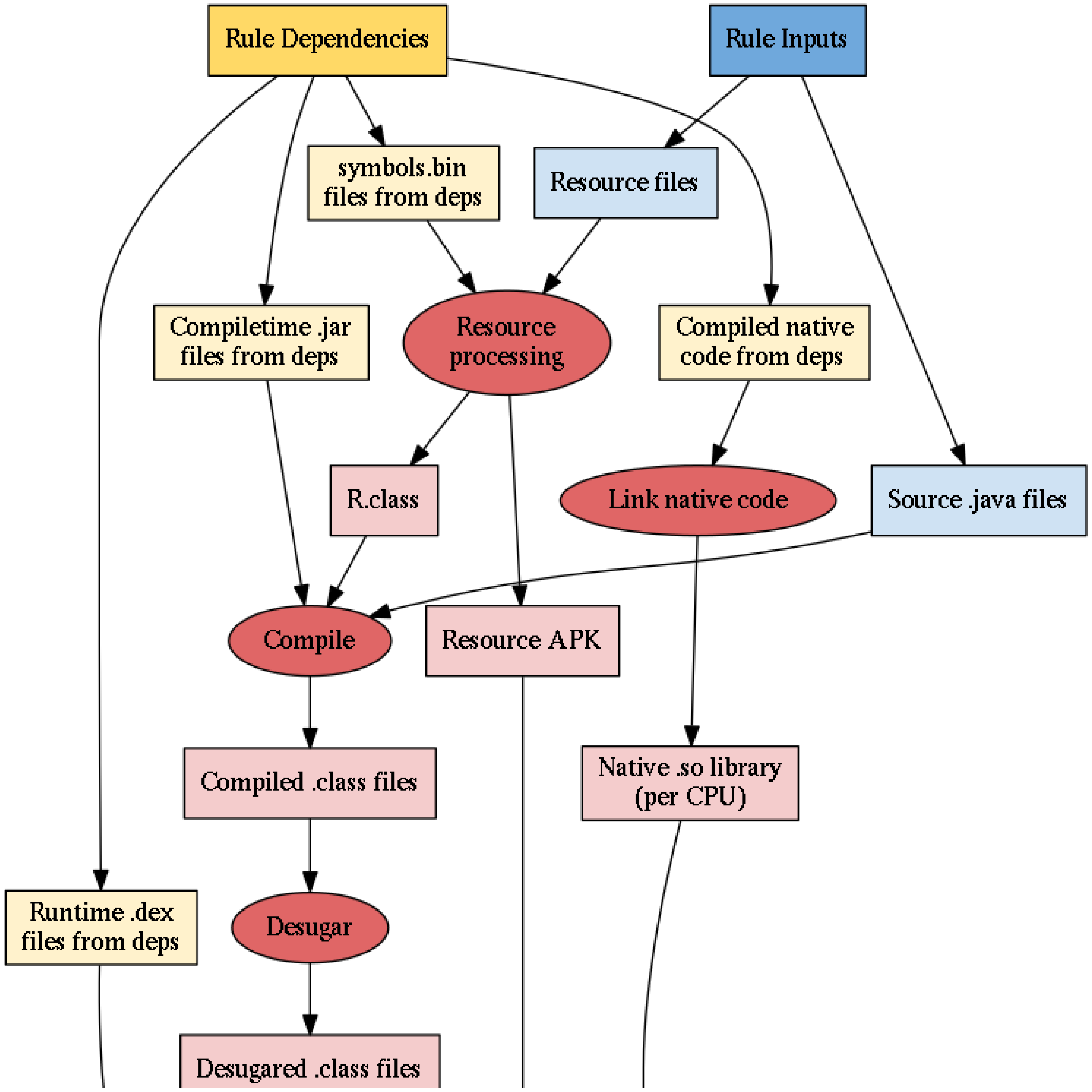
Finally, Bazel does some additional work on the compiled code:

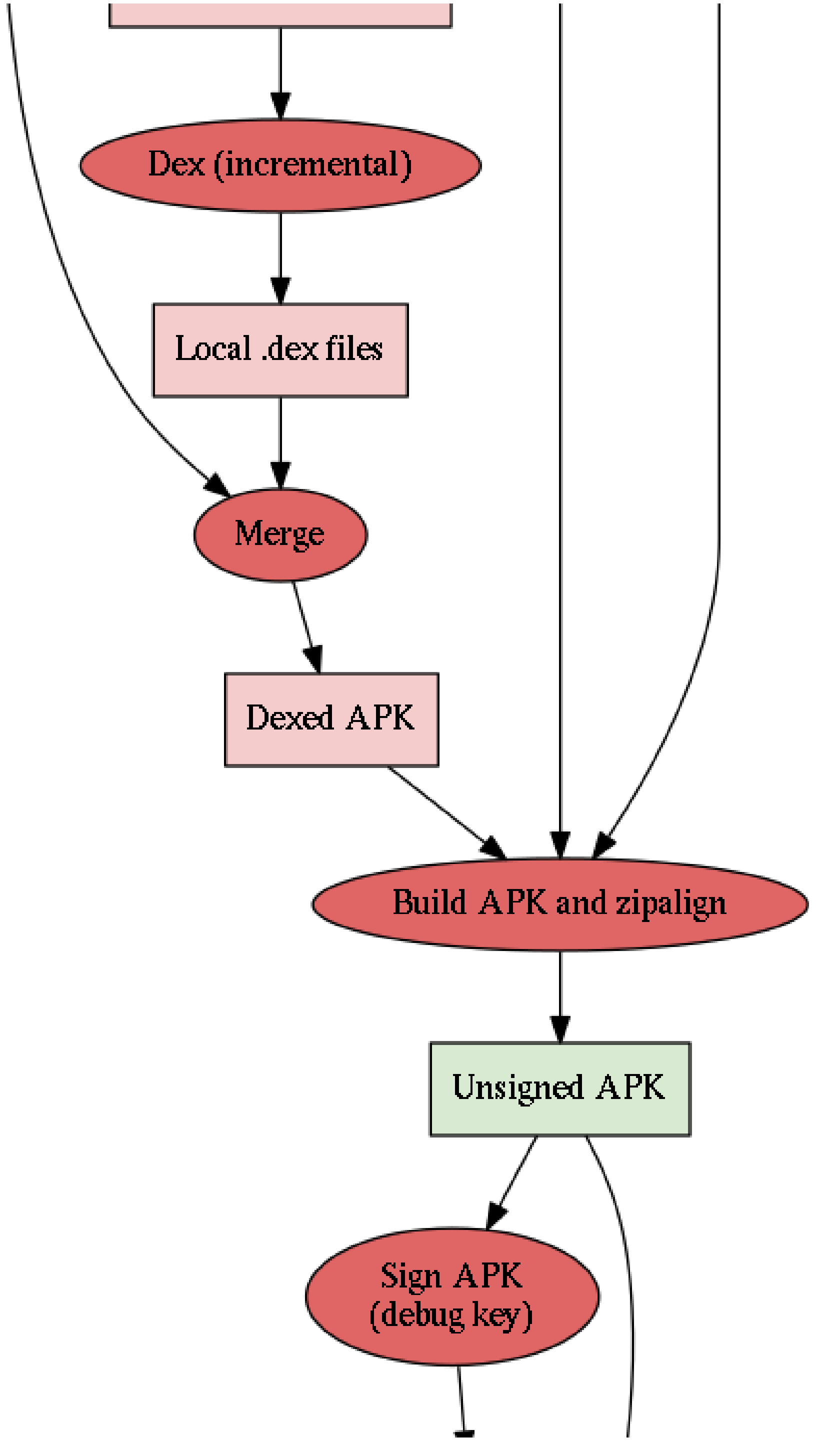
1. The compiled `.class` files are desugared (https://github.com/bazelbuild/bazel/blob/master/src/tools/android/java/com/google/devtools/build/android/desugar/Desugar.java) to replace bytecode only supported on Java 8 with Java 7 equivalents. Bazel does this so that Java 8 language features can be used for developing the app, even though the next tool, `dx` , does not support Java 8 bytecode.
2. The desugared `.class` files are converted to `.dex` files, executables for Android devices, by `dx` . These `.dex` files are then packed into the `.jar` file used at runtime. These *incremental* *`.dex`* files, produced for

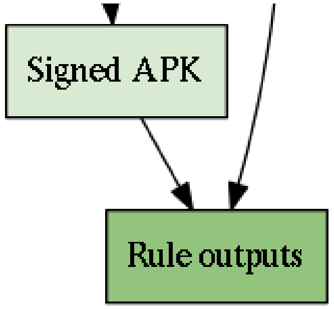
each library, mean that, when some libraries from an app are changed, only those libraries, and not the entire app, need to be re-dexed.

3. The source `.java` files for this library are used by `hjar` to generate a `jar` of `.class` files. Method bodies and private fields are removed from this compile-time `.jar`, and targets that depend on this library are compiled against this smaller `.jar`. Since these jars contain just the interface of the library, when private fields or method implementations change, dependent libraries do not need to be recompiled (they need to be recompiled only when the interface of the library changes), which results in faster builds.

Android Binaries


Rule Dependencies
Rule Inputs
symbols.bin files from deps
Resource files
Compiletime .jar files from deps
Resource processing
Compiled native code from deps
R.class
Link native code
Source .java files
Compile
Resource APK
Compiled .class files
Native .so library (per CPU)
Runtime .dex files from deps
Desugar
Desugared .class files


Dex (incremental)
Local .dex files
Merge
Dexed APK
Build APK and zipalign
Unsigned APK
Sign APK
(debug key)



An `android_binary` (https://docs.bazel.build/versions/master/be/android.html#android_binary) rule packages the entire target and its dependencies into an APK. On a high level, binaries are built similarly to libraries. However, there are a few key differences.

For binaries, the three main resource processing actions (parse, merge, and validate), are all combined into a single large action. In libraries, Java compilation can get started while validation is still ongoing, but in binaries, since we need the final resource IDs from validation, we can't take advantage of similar parallelization. Since creating more actions always introduces a small cost, and there's no parallelization available to make up for it, having a single resource processing action is actually more efficient.

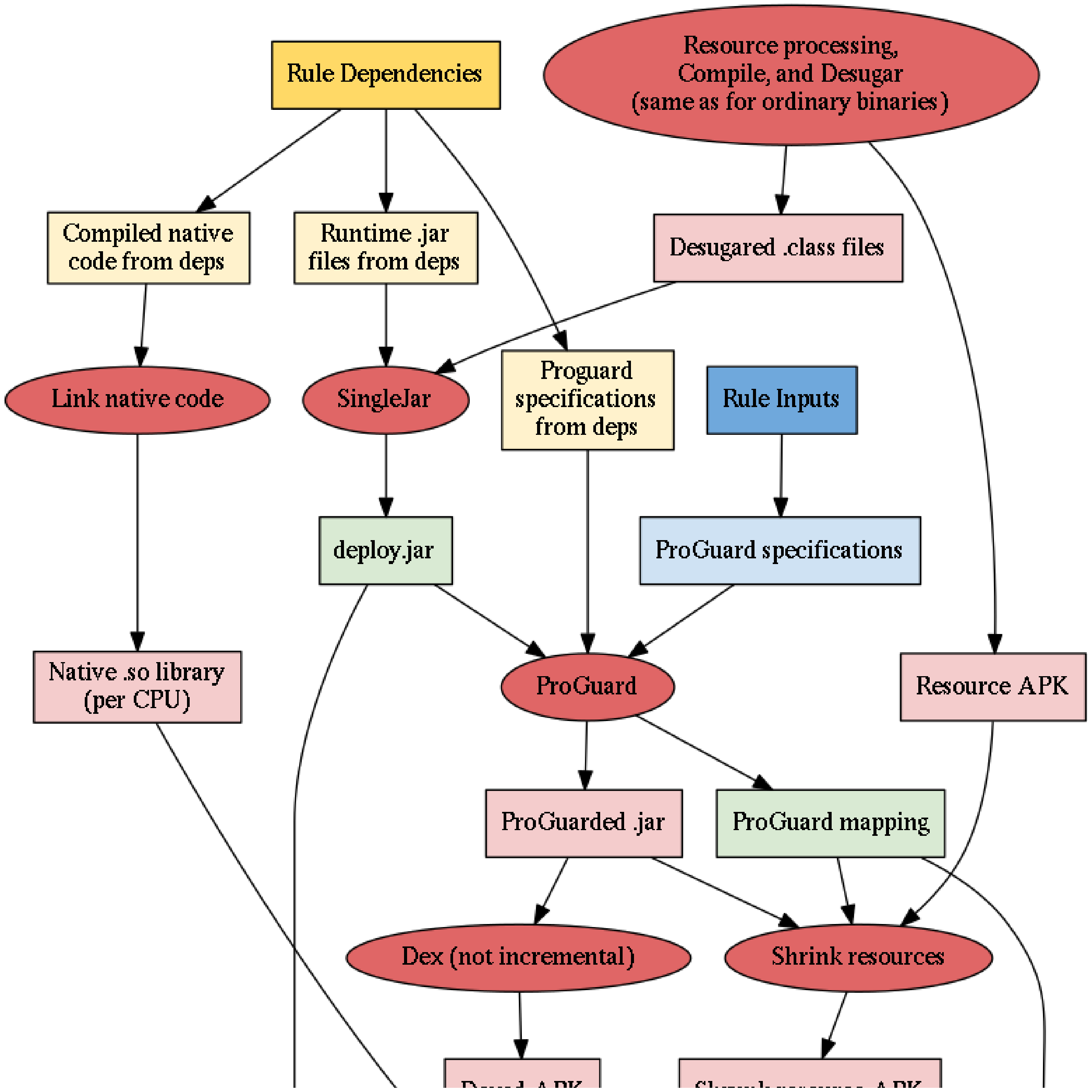
In binaries, the Java code is compiled, desugared, and dexed, just like in libraries. However, afterwards, the `.dex` files from the binary are merged together with the `.dex` files from dependencies.

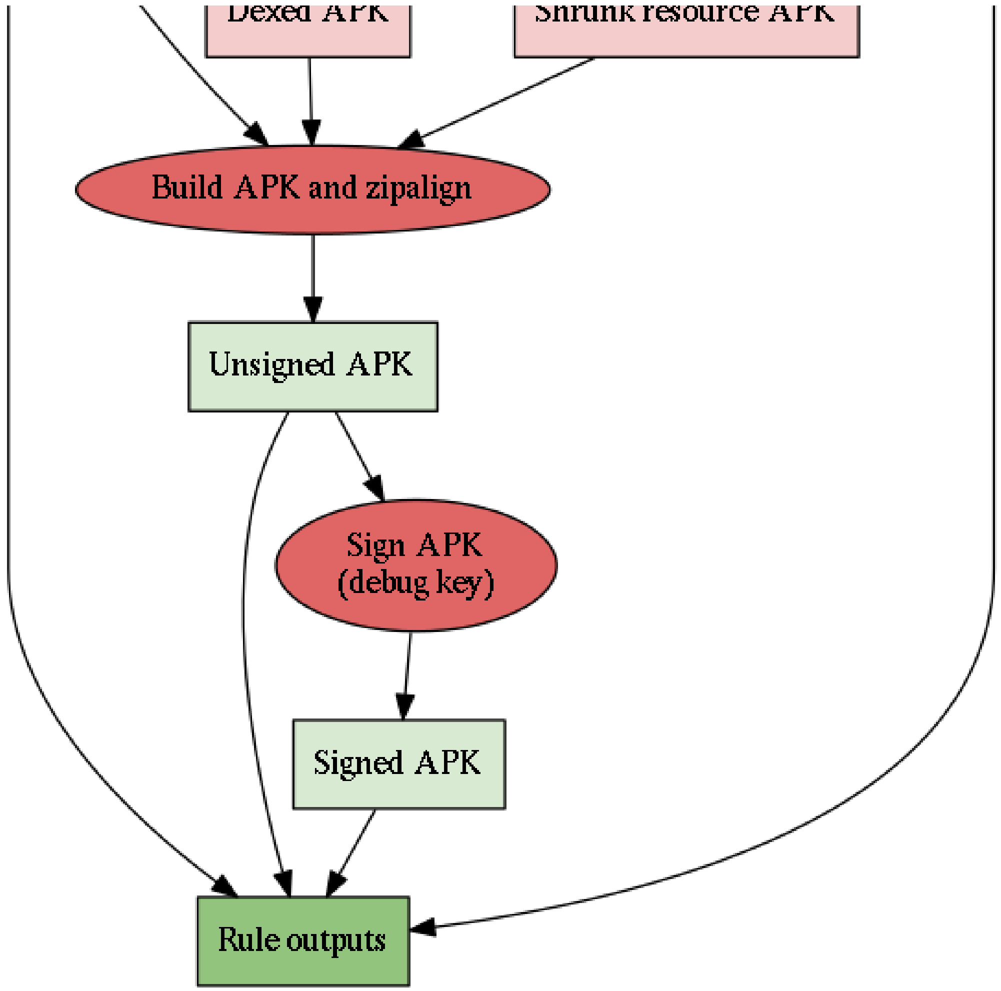
Bazel also links together compiled `C` and `C++` native code from dependencies into a single `.so` file for each CPU architecture specified by the `--fat_apk_cpu` (https://docs.bazel.build/versions/master/user-manual.html#flag--fat_apk_cpu) flag.

The merged `.dex` files, the `.so` files, and the resource APK are all combined to build an initial binary APK, which is then zipaligned (https://developer.android.com/studio/command-line/zipalign.html) to produce an unsigned APK. Finally, the unsigned APK is signed with the binary's debug key to produce a signed APK.

The merged `.dex` files are combined with the resource APK to build an initial binary APK, which is then zipaligned (https://developer.android.com/studio/command-line/zipalign.html) to produce an unsigned APK. Finally, the unsigned APK is signed with the binary's debug key to produce a signed APK.

ProGuarded Android Binaries


Rule Dependencies
Resource processing, Compile, and Desugar (same as for ordinary binaries)
Compiled native code from deps
Runtime .jar files from deps
Desugared .class files
Link native code
SingleJar
Proguard specifications from deps
Rule Inputs
deploy.jar
ProGuard specifications
Native .so library (per CPU)
ProGuard
Resource APK
ProGuarded .jar
ProGuard mapping
Dex (not incremental)
Shrink resources



Bazel supports running ProGuard (https://www.guardsquare.com/en/proguard) against `android_binary` targets to optimize them and reduce their size (https://www.guardsquare.com/en/proguard/manual/introduction). Using ProGuard substantially changes elements of the build process. In particular, the build process does not use incremental `.dex` files at all, as ProGuard can only run on `.class` files, not `.dex` files.

ProGuarding uses a `deploy.jar` file, a single `.jar` file with all of the binary's Java bytecode, created from the binary's desugared (but not dexed) `.class` files as well as the binary's transitive runtime `.jar` files. (This `deploy.jar` file is an output of all `android_binary` targets, but it doesn't play a substantial role in builds without ProGuarding.)

Based on information from a series of Proguard specifications (from both the binary and its transitive dependencies), ProGuard makes serveral passes through the `deploy.jar` in order to optimize the code, remove unused methods and fields, and shorten and obfuscate the names of the methods and fields that remain. In addition to the resulting proguarded `.jar` file, ProGuard also outputs a mapping from old to new names of methods and fields.

ProGuard's output is not dexed, so when building with ProGuard, the entire `.jar` must be re-dexed (even code from dependencies that were dexed incrementally). The dexed code is then built into the APK as usual.

ProGuard will also remove references to unused resources from the class files. If resource shrinking (https://docs.bazel.build/versions/master/be/android.html#android_binary.shrink_resources) is enabled, the resource shrinker uses the proguard output to figure out what resources are no longer used, and then uses `aapt` or `aapt2` to create a new, smaller resource APK with those resources removed. The shrunk resource APK and the dexed APK are then fed into the APK building process, which operates the same as it would without ProGuard.

Mobile-install

Mobile-install (https://docs.bazel.build/versions/master/mobile-install.html) is a way of rapidly building and deploying Android applications iteratively. It's based off of `android_binary`, but has some additional functionality to make builds and deployments more incremental.

By Alex Steinberg (https://github.com/asteinb)

About

Who's using Bazel (https://github.com/bazelbuild/bazel/wiki/Bazel-Users)

Support

Stack Overflow (https://stackoverflow.com/questions/tagged/bazel)

Issue Tracker (https://github.com/bazelbuild/bazel/issues)

Stay Connected

Twitter (https://twitter.com/bazelbuild)

Blog (https://blog.bazel.build)