Bazel

Versioned Runtime APIs under Platforms & Toolchains

Authors: schmitt@bazel.build
Reviewers:
- trybka@google.com (LGTM)
- hlopko@google.com (LGTM)
- jcater@google.com (LGTM)
- cushon@google.com (LGTM)
- cpovirk@google.com (LGTM)
- kaipi@google.com (LGTM)
- djwhang@google.com (LGTM)

Created: 2019-07-09

Please read Bazel Code of Conduct before commenting.

Overview

Most runtime libraries are versioned with a guarantee of forward API compatibility but no backwards compatibility. Examples include the Java Runtime Environment (JRE), the iOS SDK, Android SDK, libc and many more.[1] [2]

When compiling an executable that depends on such a library one has to provide the "minimum library version" to support, for example by passing that version's headers or passing an argument to a tool such as a compiler. The resulting binary will run on a system that makes that version or newer of the library available.

Platforms describe the runtime environment for binaries and as such are a logical place to contain information about the available runtime libraries and their versions. Toolchains describe a set of tools and/or libraries used by Bazel rules to execute actions such as

[1] Note that while runtime libraries are often distributed with a set of language tooling (e.g. compiler, linker) the toolings' versions do not adhere to the same system or compatibility scheme as runtime libraries. Therefore a language's tools and runtime library should be modeled in separate toolchains in Bazel or at least independently: E.g. A tool-based toolchain that has access to a range of runtime headers for its language (not just the version distributed with the current version of the tools).
[2] Yes, Python does not fall into this except schema due to Python 3 breaking backwards compatibility. The proposal below still covers its use case.

compiling a source file and are thus need to be aware of the desired minimum library versions to correctly configure action inputs and command lines.

This document explores several approaches on how to model runtime library versioning and minimum versions used during a build in platforms and toolchains while taking into account situations like creating multiple inter-dependent binaries (with different desired minimum versions) in a single build.

Background & Use Cases

Examples of where minimum runtime library versions are used and how they are modeled in Bazel today:

iOS

(Also other operating systems provided by Apple)

For iOS the compiler enforces that any APIs used are available in the requested minimum OS version (the runtime libraries are bundled and versioned with the OS). Thus the desired minimum version must be passed to each compile action and is also embedded in the compiled artifact for consistency checking at link time. iOS tooling also uses the minimum version while processing/compiling resources.

On iOS a deployable artifact (.ipa) may contain multiple binaries each of which may have a different minimum version. For example an app may be using minimum version X but come with an extension only supported at API version X+2. The extension would have its own binary using minimum version X+2 and only be used by iOS in case the current runtime supports it.

Bazel models this by offering a *minimum_os* attribute on the binary level for deployables targeting Apple OSs. This attribute is propagated via a configuration transition to all dependencies of a binary target to ensure a consistent minimum version.

Android

For Android there are several different approaches, further complicated by the fact that java and native dependencies are treated differently. Most of these mechanisms rely on flags passed to the build invocation. Because of this, targets in a single Bazel invocation must all use the same minimum version.

(Java) android_library

One way to set the minimum version of the Android SDK is to pass a specific version to *--android_sdk*. Code using newer APIs will not compile as a result. More typically however builds pass the newest available SDK version to *--android_sdk* and include appcompat as a

library that will backport newer APIs to old runtimes (using runtime checks). In effect this sets the minimum version to however far back appcompat can support the used APIs.

Separate from providing interfaces Android binary and library rules can also specify a minimum SDK version in their manifest. This number is largely ignored, however with newer manifest merging if a binary has a lower minimum version than a library it depends on an error is raised. There are plans for more API usage warnings and errors based on the manifest.

Native dependencies

Android applications can have native dependencies which depend on (libc) headers in the NDK. The NDK itself is versioned by revisions (r17, r18, r19), and each revision supports a different range of Android API levels. While the NDK is bundled with the SDK Bazel's current mechanisms for handling versioning of the two in a single build are independent.

Unlike java code, native code does not have an appcompat library and passes the minimum version to the compiler as part of the target triple. No newer APIs can be used. The minimum desired NDK version can be derived from --android_grte_top or a locally installed NDK. There is no check of this version against the application manifest's minimum version or the included java code.

Design

To ensure consistency across ecosystems in Bazel, recommend one way to communicate the desired version of libraries, via build settings. Allow platforms to define the *installed* runtime library version and check each binary's min version against it.

Requirements

- Minimum version can be set per binary, for its dependency configured target graph.
- Any target in that graph must be able to read the min version (or toolchains it uses have to).
- A platform can advertise which runtime library versions are installed on it so that the requested version can be checked against the installed version.

Proposal

The recommended approach is similar to what Apple rules do today, with the additional check against platforms.

- For each type of runtime library, define a build setting that can propagate its min version. Typically the build setting will be of type int but could also be a string.
- On each type of binary rule define an attribute for each type of runtime library that may need a min version specified. This attribute will set the corresponding build setting for the binary target's dependency graph.

    - Alternately (if there doesn't seem to be any need to ever set the build setting to different values within the same build for different binaries) it can be set on the command line and no attribute is needed.
- select() and toolchain resolution can operate on these build settings.
- Toolchains and rule implementation can read the build setting value to parametrize actions.
- Optional but recommended: Define a constraint setting with a list of values that corresponds to the build setting. This allows platforms to declare installed runtime versions. Toolchains can then raise an error if the target platform's runtime version constraint value does not fulfill the set min version.

Note: In some cases a single binary rule may need to propagate more than one min version, such as android_binary with the SDK, NDK and java versions.

Example

```
# example/java_rules/flags.bzl
IntProvider = provider(fields = ['value'])
def _impl(ctx):
    return IntProvider(value = ctx.build_setting_value)

int_flag = rule(
    implementation = _impl,
    # Only allow setting via transition
    build_setting = config.int(flag = False)
)
MAX_KNOWN_JRE = 8 # This value has to be manually updated

def in_min_jre_range(min, max):
    """Returns a tuple of minimum JRE constraint settings in range.

    Suitable for use in selects.with_or().

    Args:
      min: Minimum JRE version as integer, inclusive.
      max: Maximum JRE version as integer, exclusive.
    """
    target_format = "//example_java_rules:jre_min_version_%s"
    return tuple([target_format.format(x) for x in range(min, max)])

def at_least_min_jre(min):
    """Returns a tuple of minimum JRE constraint settings in range.

    Suitable for use in selects.with_or().

    Args:
```

```
      min: Minimum JRE version as integer, inclusive.
    """
    return in_jre_range(min, MAX_KNOWN_JRE+1)

# example/java_rules/BUILD
load("//example/java_rules:flags.bzl", "int_flag", "MAX_KNOWN_JRE")
int_flag(
    name = "jre_min_version",
    build_setting_default = "5"
)

constraint_setting(
    name = "jre_version",
    default_constraint_value = ":jre_version_8"
)

# Create a constraint value and min JRE config setting for each known
# JRE version.
[
    constraint_value(
        name = "jre_version_" + jre_version,
        constraint_setting = ":jre_version"
    )

    config_setting(
        name = "jre_min_version_" + jre_version,
        flag_values = {
            ":jre_min_version": jre_version,
        }
    )
for jre_version in range(5, MAX_KNOWN_JRE)]

# example/platforms/BUILD
platform(
    name = "old_machine",
    constraint_values = [
        "@platforms//cpu:x86_64",
        "@platforms//os:linux",
        "//example/java_rules:jre_version_5"
    ]
)

# example/java_rules/rules.bzl
load("//example/java_rules:flags.bzl", "IntProvider")
```

```
def _lib_impl(ctx):
    jre_min_version = ctx.attrs._jre_min_version[IntProvider].value
    # use jre_min_version in action creation

java_lib = rule(
    implementation = _lib_impl,
    attrs = {
        "_jre_version": attr.label(
            default=Label("//example/java_rules:jre_min_version"))
    }
)

def _min_jre_version_transition_impl(settings, attr):
    return {
        "//example/java_rules:jre_min_version": attr.min_jre_version
    }

_min_jre_version_transition = transition(
    implementation = _min_jre_version_transition_impl,
    inputs = [],
    outputs = ["//example/java_rules:jre_min_version"],
)

def _bin_impl(ctx):
    # Note: Target platform access not implemented yet.
    platform_jre =
        ctx.target_platform[Label("//example/java_rules:jre_version")]
    min_jre = ctx.attrs.min_jre_version
    if platform_jre and Int(platform_jre[-1:]) < min_jre:
        fail("Min JRE for this binary is larger than target JRE")
    # ...

java_bin = rule(
    implementation = _bin_impl,
    attrs = {
        "min_jre_version": attr.int()
        "deps": attr.label_list(cfg=_min_jre_version_transition)
    }
)

# //example/project/BUILD
load("//example/java_rules:flags.bzl", "in_jre_range", "at_least_jre")

java_lib(
    name = "lib",
```

```
    srcs = ["A.java"]
)

java_lib(
    name = "conditional_lib",
    srcs = selects.with_or({
        in_min_jre_range(5,8): ["B_Old.java"]
        at_least_min_jre(8): ["B_New.java"]
    })
)

java_bin(
    name = "old_bin",
    jre_min_version = 5,
    deps = [
        ":lib",
        ":conditional_lib",
    ],
)

java_bin(
    name = "new_bin",
    deps = [
        ":lib",
        ":conditional_lib",
    ],
)

# When running the below command, //example/project:lib will be built
# twice, once with jre_min_version=5, once with jre_min_version=8.
# :conditional_lib will be built twice as well, once with B_Old.java and
# once with B_New.java
bazel build //example/project:new_bin //example/project:old_bin

# This command should fail because the passed platform does not support
# the requested minimum JRE version. Note that the above command
# succeeds because the default platform doesn't specify any JRE version.
bazel build --platforms=//example/platforms:old_machine \
    //example/project:new_bin
```

Ranges

As illustrated in the above example ranges can occasionally be useful when dealing with versioned runtimes, such as when constructing *select()* statements based on the min

version or when registering toolchains (and indicating which should be used at which min version level).

It is recommended that each runtime library type defines its own range functions to correctly capture its semantics of version comparison.

Hermetic Runtimes

Some platforms may not have any version of a runtime library installed, or the wrong one, or a binary maintainer wants to package the runtime with the binary to make it independent of the target platform. This can be supported in a variety of ways, mostly in the toolchain definition:

- A toolchain implementation that always bundles the runtime with the binary. This implementation could be registered as default or passed as --extra_toolchains to the build.
- A toolchain implementation can conditionally bundle the runtime with the binary if the target platform has no registered version of the runtime. Such behavior could be implemented via a select() on the platform's values that chooses the correct toolchain implementation.
- A toolchain implementation can conditionally bundle the runtime with the binary if the target platform only supports an incompatible version of the runtime, for example a version lower than the min version passed.

Library Owners

Library owners (such as for `:conditional_lib` in the example above) will want to make sure in an automated fashion that their code continues working under different min version flag values.

Testing

The below example introduces a new rule type `flag_value_test_suite` that internally split transitions for each passed value of each flag (exploring the full set of combinations if there's more than one flag) and executes the given tests in each.

```
java_test(
    name = "conditional_lib_test",
    srcs = [ "Test.java" ]
)

flag_value_test_suite(
    name = "conditional_lib_jre_tests",
    tests = [ ":conditional_lib_test" ],
    flags = {
        "//example/java_rules:jre_min_version": [ "7", "8" ]
```

```
    }
)
```

Warning

Library owners may not control all dependencies on their targets or the minimum versions people may use their library with. If they want to raise a warning when their library is not compatible with the current build they can employ a select statement with error branch:

```
java_lib(
     name = "foo",
     srcs = selects.with_or(
         input_dict = { at_least_min_jre(8): ["Foo.java"] },
         no_match_error = "requires at least Java 8"
    ),
)
```

Implementation

Most of the proposed approach works with today's technology. To support min version checking against the target platform however we need to add the ability to **read a given constraint setting's value(s) from the current target platform**.

The logic for creating, selecting and verifying minimum versions and target versions will likely be similar across many rule sets and can be **bundled in a common location** (for example skylib).

`flag_value_test_suite` does not exist yet but is needed for many use cases now that configurations are proliferating in builds. It may require some additional attributes such as the configuration fragments of passed flags when it is implemented.

Alternatives Considered

Don't change anything

Semantic considerations aside (if platforms represent a binary's runtime environment then installed runtime libraries really should belong in there) we could leave each ecosystem to handle minimum versions as they (already) see fit.

Pros

- No additional infrastructure complexity
- Allows for alignment with the toolchain's ecosystem best practices outside Bazel

Cons

- Inconsistencies and confusion across different ecosystems used in Bazel (especially if they have common dependencies like native code)
- Reduced/no benefits from Bazel infrastructure like selects and toolchain selection

Synthetic Platforms

Similar to the build setting proposal in the design section, however min versions are propagated in the platform (via configuration values) rather than build settings.

- Runtime versions are available as constraint values
- Binary target refers to min version constraint value, this is then merged into set target platform(s)
    - implies same deployable could contain multiple binaries each targeting a different platform (which is already true for multi-arch deployables)
- Selects, toolchains operate on platform constraint values
    - probably need some kind of range function (toolchain supports version X through Y) which implies ordering on constraint values
- Toolchain resolution (or toolchain directly) reads runtime version from platform

Cons

- Makes platforms more difficult to understand.
- Cannot check values against platform installed runtime.