BUILD phases

Loading Phase:

- Load and evaluate all extensions and BUILD files needed for the build.
- Instantiates rules, adding them to a graph.
- Macros are evaluated during this phase.

Analysis Phase:

- Execute the code of the rules (their implementation function).
- Instantiate actions, which describe how to generate outputs from inputs.
- Explicitly list files to be generated before executing commands.
- Generates an action graph based on the graph from the loading phase.

Execution Phase:

- Execute actions when at least one of their outputs is required.
- If a file is missing or a command fails to generate an output, the build fails.
- Tests are run during this phase.

Rules

A rule defines a series of actions that Bazel performs on inputs to produce a set of outputs, which are referenced in providers returned by the rule's implementation function. For example, a C++ binary rule might:

- Take a set of .cpp source files (inputs).
- Run g++ on the source files (action).
- Return the DefaultInfo provider with the executable output and other files to make available at runtime.
- Return the CcInfo provider with C++-specific information gathered from the target and its dependencies.

From Bazel's perspective, g++ and the standard C++ libraries are also inputs to this rule.

As a rule writer, you must consider not only the user-provided inputs to a rule, but also all of the tools and libraries required to execute the actions.

GenRules

- A few rules are built into Bazel itself. These native rules, such as genrule and filegroup, provide some core support.
- By defining your own rules, you can add support for languages and tools that Bazel doesn't support natively.
- A genrule generates one or more files using a user-defined Bash command.

#genrule for concatenation of files
# Create a folder “genrules”, and add the following code in BUILD file in genrules

```
genrule(
        name = "concat",
        srcs = ["f2.txt", "f1.txt", "f3.txt"],
        outs= ["concat.txt"],
        #cmd = "cat $(location //:f1.txt) $(location //:f2.txt) > $@",
        cmd= "cat $(SRCS) > $@",

    )
```

Run the target : $ bazel build //:concat

GenRules

#genrule for processing data
# Create a folder “genrules”, and add the following code in BUILD file in genrules

```
genrule(
    name = "process_data",
    srcs = ["process_data.py", "data1.txt", "data2.txt"],
    outs = ["output.txt"],
    cmd = "python3 process_data.py data1.txt data2.txt $@",

)
```

Run the target : $ bazel build //:process_data

#genrule for copying files
# Create a folder “genrules”, and add the following code in BUILD file in genrules

```
genrule(
    name = "copy_files",
    srcs = ["data1.txt"],
    outs = ["copy_file.txt"],
    cmd = "cp $< $@",
)
```

Run the target : $ bazel build //:copy_files

GenRules

```
#process_data.py
import sys

def process_data(file1, file2, output_file):
    # Logic to process data from file1 and file2 and write to output_file
    with open(file1, 'r') as f1, open(file2, 'r') as f2, open(output_file, 'w') as out:
        data1 = f1.read()
        data2 = f2.read()
        out.write(data1 + data2)

if __name__ == "__main__":
    if len(sys.argv) != 4:
        print("Usage: python process_data.py <input_file1> <input_file2> <output_file>")
        sys.exit(1)

    input_file1 = sys.argv[1]
    input_file2 = sys.argv[2]
    output_file = sys.argv[3]

    process_data(input_file1, input_file2, output_file)
```

filegroup

- Use filegroup to assign a convenient name to a collection of targets.
- These named collections can be referenced from other rules within the build system.
- It's encouraged to use filegroup instead of directly referencing directories.
- Directly referencing directories is unsound because the build system may not have full knowledge of all files within the directory, leading to potential issues where it fails to rebuild when files change.
- When combined with glob, filegroup ensures that all files are explicitly known to the build system, enhancing build stability and reliability.

Genrules with filegroup

```
filegroup(
    name = "files",
    srcs = ["f1.txt", "f2.txt", "f3.txt"],
)

genrule(
    name = "cat_files",
    srcs = [":files", ":copy_files"],
    outs = ["cat_files.txt"],
    cmd =  "cat $(locations :files) $(location :copy_files)> $@",

)
```

Customised rules

Already available rules
https://bazel.build/rules

```
# rule to generate a file (foo.bzl)

def _foo_binary_impl(ctx):
        out = ctx.actions.declare_file(ctx.label.name)
        ctx.actions.write(
            output = out,
            content = "Hello!\n",
        )
        return [DefaultInfo(files = depset([out]))]

foo_binary = rule(
    implementation = _foo_binary_impl,
)
```

```
#BUILD file

load("//:foo.bzl", "foo_binary")
print("BUILD file")

foo_binary(name = "bin1")
foo_binary(name = "bin2")
```

Customised rules

```
# rule to generate a file (foo1.bzl)

def _foo_binary1_impl(ctx):
    out = ctx.actions.declare_file(ctx.label.name)
    ctx.actions.write(
        output = out,
        content = "Hello {}!\n".format(ctx.attr.username),
    )
    return [DefaultInfo(files = depset([out]))]

foo_binary1 = rule(
    implementation = _foo_binary1_impl,
    attrs = {
        "username": attr.string(),
    },
))
```

```
#BUILD file

load("//:foo.bzl", "foo_binary")
load("//:foo1.bzl", "foo_binary1")

print("BUILD file")

foo_binary(name = "bin1")
foo_binary(name = "bin2")
foo_binary1(
    name = "bin",
    username = "Alice",
)
```

Customised rules

#hello-world.bzl

```
def _hello_world_impl(ctx):
    out = ctx.actions.declare_file(ctx.label.name + ".cc")
    ctx.actions.expand_template(
        output = out,
        template = ctx.file.template,
        substitutions = {"{NAME}": ctx.attr.username},
    )
    return [DefaultInfo(files = depset([out]))]

hello_world = rule(
    implementation = _hello_world_impl,
    attrs = {
        "username": attr.string(default = "unknown person"),
        "template": attr.label(
            allow_single_file = [".cc.tpl"],
            mandatory = True,
        ),
    },
)
```

//file.cc.tpl

```
#include <iostream>

int main() {
    std::cout << "Hello, {NAME}!\n";
    return 0;
}
```

#BUILD

load("//:hello-world.bzl", "hello_world")

```
hello_world(
    name = "hello",
    username = "Alice",
    template = "file.cc.tpl",
)
cc_binary(
    name = "hello_bin",
    srcs = [":hello"],
)
```