just, a build system

Dept: Intelligent Cloud Technologies Lab, Huawei Munich Research Center
Date: Fall 2022

HUAWEI

Build Systems (Overview)

A build system …

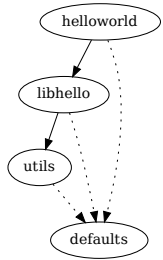
- computes a *function*
  - from source files to final artifacts (libraries, binaries, etc)
  - typically composed of smaller functions (“actions”) like individual compiler invocations
- … that is declared in terms meaningful to a programmer
  - like “library”, “binary”; *not* individual object files, etc
  - without hard-coding language-specific knowledge ($\rightarrow$ user-defined “rules”)
  - allowing multi-language builds
- Requirements
  - must be correct
  - should be fast

just Example

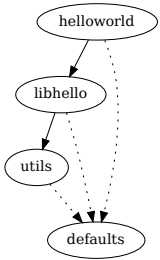
```
$ cat TARGETS
{ "helloworld":
  { "type": ["@", "rules", "CC", "binary"]
  , "name": ["helloworld"]
  , "srcs": ["main.cpp"]
  , "deps": ["libhello"]
  }
, "libhello":
  { "type": ["@", "rules", "CC", "library"]
  , "name": ["hello"]
  , "srcs": ["hello.cpp"]
  , "hdrs": ["hello.hpp"]
  , "deps": ["utils"]
  }
, "utils":
  { "type": ["@", "rules", "CC", "library"]
  , "name": ["utils"]
  , "srcs": ["utils.cpp"]
  , "hdrs": ["utils.hpp"]
  }
}
$
```

just Example

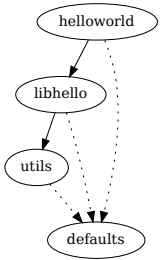
```
$ cat TARGETS
{ "helloworld":
  { "type": ["@", "rules", "CC", "binary"]
  , "name": ["helloworld"]
  , "srcs": ["main.cpp"]
  , "deps": ["libhello"]
  }
, "libhello":
  { "type": ["@", "rules", "CC", "library"]
  , "name": ["hello"]
  , "srcs": ["hello.cpp"]
  , "hdrs": ["hello.hpp"]
  , "deps": ["utils"]
  }
, "utils":
  { "type": ["@", "rules", "CC", "library"]
  , "name": ["utils"]
  , "srcs": ["utils.cpp"]
  , "hdrs": ["utils.hpp"]
  }
}
$
```

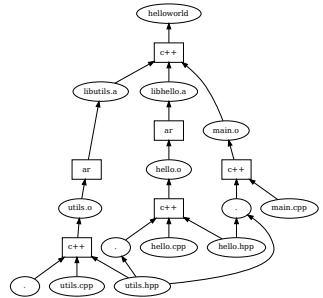


just Example

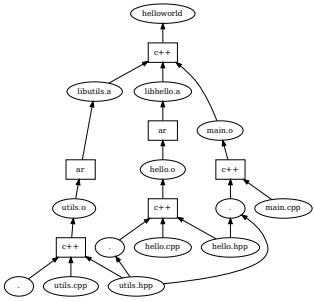


just Example

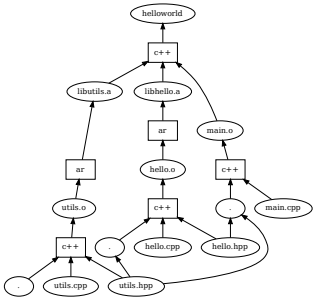




just Example

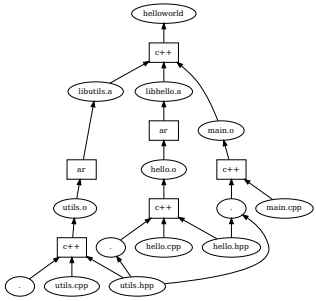


just Example




```
$ just build -C repos.json helloworld
INFO: Requested target is [["@","","","helloworld"],{}]
INFO: Analysed target [["@","","","helloworld"],{}]
INFO: Export targets found: 0 cached, 0 uncached, 0 not eligible for caching
INFO: Discovered 6 actions, 3 trees, 0 blobs
INFO: Building [["@","","","helloworld"],{}].
INFO: Processed 6 actions, 0 cache hits.
INFO: Artifacts built, logical paths are:
       helloworld [70020e8a5004bf6fa2f91fbb2cddca476e7723f5:18448:x]
$
```

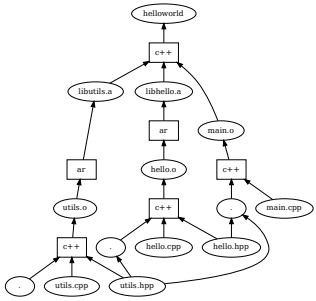
just Example




```
$ just build -C repos.json helloworld
INFO: Requested target is [["@","","","helloworld"],{}]
INFO: Analysed target [["@","","","helloworld"],{}]
INFO: Export targets found: 0 cached, 0 uncached, 0 not eligible for caching
INFO: Discovered 6 actions, 3 trees, 0 blobs
INFO: Building [["@","","","helloworld"],{}].
INFO: Processed 6 actions, 0 cache hits.
INFO: Artifacts built, logical paths are:
       helloworld [70020e8a5004bf6fa2f91fbb2cddca476e7723f5:18448:x]
$

$ just install -C repos.json -o . helloworld
INFO: Requested target is [["@","","","helloworld"],{}]
INFO: Analysed target [["@","","","helloworld"],{}]
INFO: Export targets found: 0 cached, 0 uncached, 0 not eligible for caching
INFO: Discovered 6 actions, 3 trees, 0 blobs
INFO: Building [["@","","","helloworld"],{}].
INFO: Processed 6 actions, 6 cache hits.
INFO: Artifacts can be found in:
     /worker/build/62b481553d2fe448/root/work/helloworld/helloworld [70020e8a5004bf6fa2f91fbb2cddca476e7723f5:184
$
```

just Example




```
$ just build -C repos.json helloworld
INFO: Requested target is [["@","","","helloworld"],{}]
INFO: Analysed target [["@","","","helloworld"],{}]
INFO: Export targets found: 0 cached, 0 uncached, 0 not eligible for caching
INFO: Discovered 6 actions, 3 trees, 0 blobs
INFO: Building [["@","","","helloworld"],{}].
INFO: Processed 6 actions, 0 cache hits.
INFO: Artifacts built, logical paths are:
       helloworld [70020e8a5004bf6fa2f91fbb2cddca476e7723f5:18448:x]
$

$ just install -C repos.json -o . helloworld
INFO: Requested target is [["@","","","helloworld"],{}]
INFO: Analysed target [["@","","","helloworld"],{}]
INFO: Export targets found: 0 cached, 0 uncached, 0 not eligible for caching
INFO: Discovered 6 actions, 3 trees, 0 blobs
INFO: Building [["@","","","helloworld"],{}].
INFO: Processed 6 actions, 6 cache hits.
INFO: Artifacts can be found in:
     /worker/build/62b481553d2fe448/root/work/helloworld/helloworld [70020e8a5004bf6fa2f91fbb2cddca476e7723f5:184
$

$ ./helloworld
Hello World!
$
```

Remote Build Execution

A remote build execution system consists of

Remote Build Execution

A remote build execution system consists of

- a Content-Adressable Store (CAS)
  (files, indexed by (essentially) their hash)

Remote Build Execution

A remote build execution system consists of

- a Content-Adressable Store (CAS)
  (files, indexed by (essentially) their hash)
- the actual execution service

Remote Build Execution

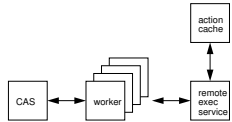
A remote build execution system consists of

- a Content-Adressable Store (CAS)
  (files, indexed by (essentially) their hash)
- the actual execution service
  - using many workers, sharing files via the CAS

Remote Build Execution

A remote build execution system consists of

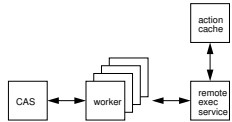
- a Content-Adressable Store (CAS)
  (files, indexed by (essentially) their hash)
- the actual execution service
  - using many workers, sharing files via the CAS
  - using an action cache (AC)



Remote Build Execution

A remote build execution system consists of

- a Content-Adressable Store (CAS)
- the actual execution service



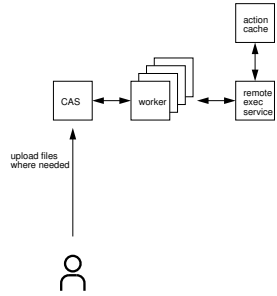
Remote Build Execution

A remote build execution system consists of

- a Content-Adressable Store (CAS)
- the actual execution service

To execute an action

- files unknown to the CAS are uploaded



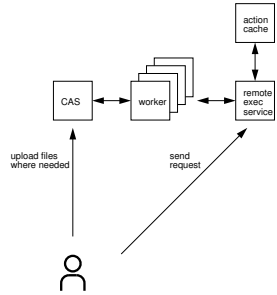
Remote Build Execution

A remote build execution system consists of

- a Content-Adressable Store (CAS)
- the actual execution service

To execute an action

- files unknown to the CAS are uploaded
- the action is requested



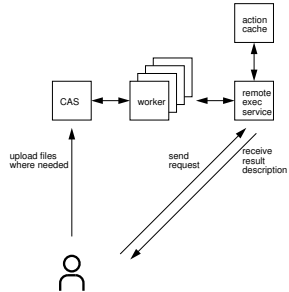
Remote Build Execution

A remote build execution system consists of

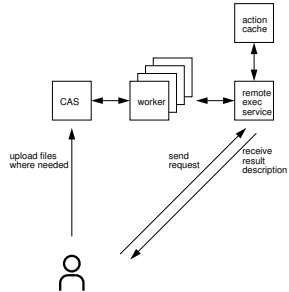
- a Content-Adressable Store (CAS)
- the actual execution service

To execute an action

- files unknown to the CAS are uploaded
- the action is requested
- a description of the output is received, typically from AC



Remote Build Execution



A remote build execution system consists of

- a Content-Adressable Store (CAS)
- the actual execution service

To execute an action

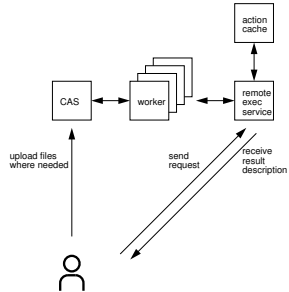
- files unknown to the CAS are uploaded
- the action is requested
- a description of the output is received, typically from AC
- actual artifacts can be downloaded from CAS, should they be needed

Remote Build Execution

A remote build execution system consists of

- a Content-Adressable Store (CAS)
- the actual execution service

To execute an action upload, request, receive answer

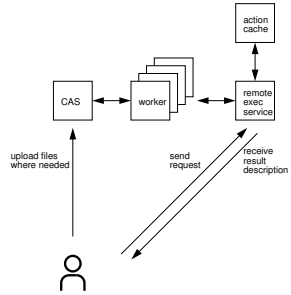


Remote Build Execution

A remote build execution system consists of

- a Content-Adressable Store (CAS)
- the actual execution service

To execute an action upload, request, receive answer



Benefits of remote execution

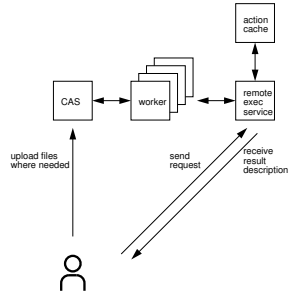
- every action executed in isolation; dependencies are correct
- AC can be shared between developpers
- better parallelism

Remote Build Execution

A ~~remote~~ build execution system consists of

- a Content-Adressable Store (CAS)
- the actual execution service

To execute an action upload, request, receive answer



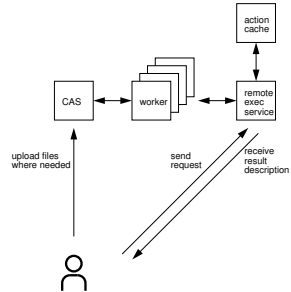
Benefits of remote execution

- every action executed in isolation; dependencies are correct
- AC can be shared between developpers
- better parallelism

But also works locally!

⇝ actions can have their own view and output convention *(conflict-free by design)*

Remote Build Execution



A ~~remote~~ build execution system consists of

- a Content-Adressable Store (CAS)
- the actual execution service

To execute an action upload, request, receive answer

Benefits of remote execution

- every action executed in isolation; dependencies are correct
- AC can be shared between developpers
- better parallelism

But also works locally!

⇝ actions can have their own view and output convention *(conflict-free by design)*

As people use `git` as VCS, let's use `git` blob/tree identifiers everywhere!

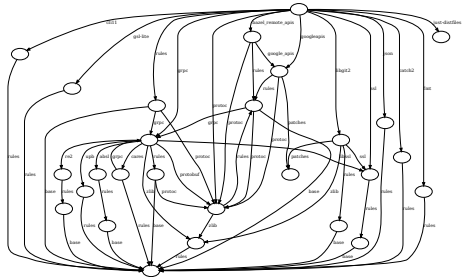
Multi-Repository Builds

- Code can be split over many repositories
  (also good do avoid duplication, e.g., rules)

Multi-Repository Builds

- Code can be split over many repositories
  (also good do avoid duplication, e.g., rules)
- Have to refer to other repositories
  - agreeing on global names doesn't work
  - often "any `libfoo` will do"

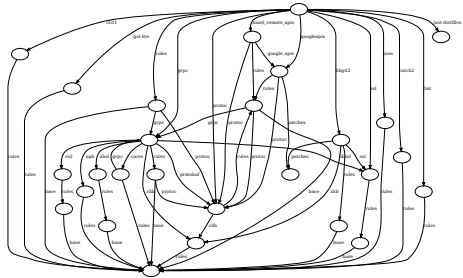
Multi-Repository Builds



- Code can be split over many repositories *(also good do avoid duplication, e.g., rules)*
- Have to refer to other repositories
  - agreeing on global names doesn't work
  - often "any `libfoo` will do"

⇝ use local names and bind in a project configuration (get DFA)

Multi-Repository Builds

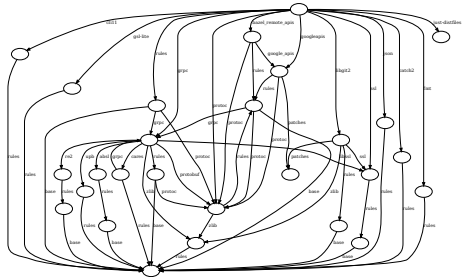


- Code can be split over many repositories
  (also good do avoid duplication, e.g., rules)
- Have to refer to other repositories
  - agreeing on global names doesn't work
  - often "any `libfoo` will do"

  ⇝ use local names and bind in a project configuration (get DFA)
- Repo semantics must be independent of caller—or if "main" repository

  ⇝ let targets decide where to logically place artifacts; we have staging anyway

Multi-Repository Builds

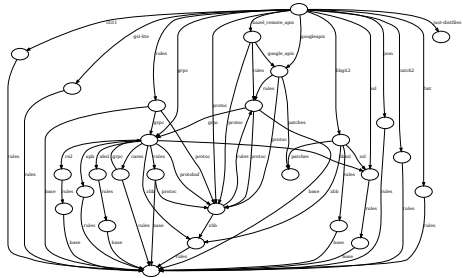


- Code can be split over many repositories
  (also good do avoid duplication, e.g., rules)
- Have to refer to other repositories
  - agreeing on global names doesn't work
  - often "any `libfoo` will do"

  ⇝ use local names and bind in a project configuration (get DFA)
- Repo semantics must be independent of caller—or if "main" repository

  ⇝ let targets decide where to logically place artifacts; we have staging anyway
- As location doesn't matter, can as well use `git` trees as roots ⇝ quickly get blob ids
  (and use one default repository to store everything)

Multi-Repository Builds



- Code can be split over many repositories
  (also good do avoid duplication, e.g., rules)
- Have to refer to other repositories
  - agreeing on global names doesn't work
  - often "any `libfoo` will do"

  ⇝ use local names and bind in a project configuration (get DFA)
- Repo semantics must be independent of caller—or if "main" repository
  ⇝ let targets decide where to logically place artifacts; we have staging anyway
- As location doesn't matter, can as well use `git` trees as roots ⇝ quickly get blob ids
  (and use one default repository to store everything)
- Additional benefit: target-level caching if reachable part of DFA unchanged
  (minimal DFA as canonical representation; plus target name, configuration)
  ⇝ keep graphs to handle small; still flexible to build in different configurations

Layers

- From a repository, different information is taken
  - source files
  - description of the targets
  - definition of the rules and their expressions

Layers

- From a repository, different information is taken
  - source files
  - description of the targets
  - definition of the rules and their expressions
- No need that they all come from the same file root!
  - separate source/target roots useful for building third-party software
  - … or even for just picking up preinstalled dependencies

Layers

- From a repository, different information is taken
  - source files
  - description of the targets
  - definition of the rules and their expressions
- No need that they all come from the same file root!
  - separate source/target roots useful for building third-party software
  - … or even for just picking up preinstalled dependencies
- Nothing special about the name `TARGETS` either …

Layers Example: Building Third-Party Code

```
$ cat repos.json
{ "repositories":
  { "":
    { "workspace_root": ["file", "third_party/helloworld"]
    , "target_root": ["file", "etc/imports"]
    , "target_file_name": "TARGETS.hello"
    , "bindings": {"rules": "rules", "patches": "patches"}
    }
  , "patches": {"workspace_root": ["file", "patches"]}
  , "rules": {"workspace_root": ["file", "../rules"]}
  }
}
$
```

Layers Example: Building Third-Party Code

```
$ ls third_party/helloworld
hello.cpp
hello.hpp
main.cpp
utils.cpp
utils.hpp
$
```

Layers Example: Building Third-Party Code

```
$ cat etc/imports/TARGETS.hello
{ "helloworld":
  { "type": ["@", "rules", "CC", "binary"]
  , "name": ["helloworld"]
  , "srcs": ["main.cpp"]
  , "deps": ["libhello"]
  }
, "libhello":
  { "type": ["@", "rules", "CC", "library"]
  , "name": ["hello"]
  , "srcs": ["hello.cpp"]
  , "hdrs": ["hello.hpp"]
  , "deps": ["utils"]
  }
, "utils":
  { "type": ["@", "rules", "CC", "library"]
  , "name": ["utils"]
  , "srcs": ["utils.cpp"]
  , "hdrs": ["utils.hpp"]
  }
}
$
```

Layers Example: Building Third-Party Code

```
$ just build -C repos.json helloworld
INFO: Requested target is [["@","","","helloworld"],{}]
INFO: Analysed target [["@","","","helloworld"],{}]
INFO: Export targets found: 0 cached, 0 uncached, 0 not eligible for caching
INFO: Discovered 6 actions, 3 trees, 0 blobs
INFO: Building [["@","","","helloworld"],{}].
INFO: Processed 6 actions, 0 cache hits.
INFO: Artifacts built, logical paths are:
        helloworld [70020e8a5004bf6fa2f91fbb2cddca476e7723f5:18448:x]
$
```

Layers Example: Building Third-Party Code

```
$ just install -C repos.json -o . main.cpp && cp main.cpp main.cpp.orig \
  && (echo '%s/World/Universe/'; echo 'w'; echo 'q') | ed main.cpp \
  && (diff -u main.cpp.orig main.cpp > patches/main.diff || :) && rm main.cpp*
INFO: Requested target is [["@","","","main.cpp"],{}]
INFO: Analysed target [["@","","","main.cpp"],{}]
INFO: Export targets found: 0 cached, 0 uncached, 0 not eligible for caching
INFO: Discovered 0 actions, 0 trees, 0 blobs
INFO: Building [["@","","","main.cpp"],{}].
INFO: Processed 0 actions, 0 cache hits.
INFO: Artifacts can be found in:
       /worker/build/626f7f7e70da0cb1/root/work/myproject/main.cpp [93fa74581864061cace4a388a66ebaafaa823f81:88:f]
88
91
$
```

Layers Example: Building Third-Party Code

Interlude: entity naming

- single string: `"foo"`
  target in the same module (i.e., directory)
- list of length 2: `["foo", "bar"]`
  module and target
- list of length ≥ 3: first entry determines naming scheme
  - `["@", local repo name, module, target]`

If the target is not explicitly declared, fall back to source file of that name

Layers Example: Building Third-Party Code

Interlude: entity naming

- single string: `"foo"`
  target in the same module (i.e., directory)
- list of length 2: `["foo", "bar"]`
  module and target
- list of length ≥ 3: first entry determines naming scheme
  - `["@", local repo name, module, target]`
  - `["./", relative module path, target]`
  - `["FILE", null, name]`
    explicitly a file in the current module
  - `["TREE", null, name]`
    explicitly the directory in the current module, rooted at the given name
  - ...

If the target is not explicitly declared, fall back to source file of that name

Layers Example: Building Third-Party Code

```
$ ${EDITOR} etc/imports/TARGETS.hello && cat etc/imports/TARGETS.hello
{ "helloworld":
  { "type": ["@", "rules", "CC", "binary"]
  , "name": ["helloworld"]
  , "srcs": ["main.cpp"]
  , "deps": ["libhello"]
  }
, "libhello":
  { "type": ["@", "rules", "CC", "library"]
  , "name": ["hello"]
  , "srcs": ["hello.cpp"]
  , "hdrs": ["hello.hpp"]
  , "deps": ["utils"]
  }
, "utils":
  { "type": ["@", "rules", "CC", "library"]
  , "name": ["utils"]
  , "srcs": ["utils.cpp"]
  , "hdrs": ["utils.hpp"]
  }
, "main.cpp":
  { "type": ["@", "rules", "patch", "file"]
  , "src": [["FILE", null, "main.cpp"]]
  , "patch": [["@", "patches", "", "main.diff"]]
  }
}
$
```

Layers Example: Building Third-Party Code

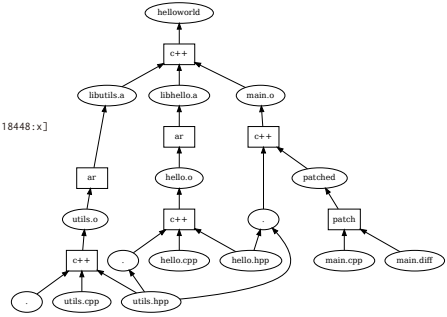
```
$ cat etc/imports/TARGETS.hello && just analyse -C repos.json --dump-actions - main.cpp
{ "helloworld":
  { "type": ["@", "rules", "CC", "binary"]
  , "name": ["helloworld"]
  , "srcs": ["main.cpp"]
  , "deps": ["libhello"]
  }
, "libhello":
  { "type": ["@", "rules", "CC", "library"]
  , "name": ["hello"]
  , "srcs": ["hello.cpp"]
  , "hdrs": ["hello.hpp"]
  , "deps": ["utils"]
  }
, "utils":
  { "type": ["@", "rules", "CC", "library"]
  , "name": ["utils"]
  , "srcs": ["utils.cpp"]
  , "hdrs": ["utils.hpp"]
  }
, "main.cpp":
  { "type": ["@", "rules", "patch", "file"]
  , "src": [["FILE", null, "main.cpp"]]
  , "patch": [["@", "patches", "", "main.diff"]]
  }
}
INFO: Requested target is [["@","","","main.cpp"],{}]
INFO: Result of target [["@","","","main.cpp"],{}]: {
        "artifacts": {
          "main.cpp": {"data":{"id":"639b4a9026450069674242e821a37d4dd2755f52","path":"patched"},"type":"ACTION"}
        },
        "provides": {
        },
        "runfiles": {
          "main.cpp": {"data":{"id":"639b4a9026450069674242e821a37d4dd2755f52","path":"patched"},"typ
        }
      }
INFO: Actions for target [["@","","","main.cpp"],{}]:
[
  {
    "command": ["patch","-s","--read-only=ignore","--follow-symlinks","-o","patched","orig","patch"]
    "input": {
      "orig": {
        "data": {
          "path": "main.cpp",
          "repository": ""
        },
        "type": "LOCAL"
      },
      "patch": {
        "data": {
          "path": "main.diff",
          "repository": "patches"
        },
        "type": "LOCAL"
      }
    },
    "output": ["patched"]
  }
]
$
```

Layers Example: Building Third-Party Code

```
$ just install -C repos.json -o . helloworld && ./helloworld
INFO: Requested target is [["@","","","helloworld"],{}]
INFO: Analysed target [["@","","","helloworld"],{}]
INFO: Export targets found: 0 cached, 0 uncached, 0 not eligible for caching
INFO: Discovered 7 actions, 3 trees, 1 blobs
INFO: Building [["@","","","helloworld"],{}].
INFO: Processed 7 actions, 4 cache hits.
INFO: Artifacts can be found in:
        /worker/build/626f7f7e70da0cb1/root/work/myproject/helloworld [8ae345e57b482a92068c4afb33dbcf5d1fd77960:18448:x]
Hello Universe!
$
```

Layers Example: Building Third-Party Code

```
$ just install -C repos.json -o . helloworld && ./helloworld
INFO: Requested target is [["@","","","helloworld"],{}]
INFO: Analysed target [["@","","","helloworld"],{}]
INFO: Export targets found: 0 cached, 0 uncached, 0 not eligible for caching
INFO: Discovered 7 actions, 3 trees, 1 blobs
INFO: Building [["@","","","helloworld"],{}].
INFO: Processed 7 actions, 4 cache hits.
INFO: Artifacts can be found in:
        /worker/build/626f7f7e70da0cb1/root/work/myproject/helloworld [8ae345e57b482a92068c4afb33dbcf5d1fd77960:18448:x]
Hello Universe!
$
```



Rules: Data of a Target

- Rules are used to describe targets of a given type, like a C++ library

Rules: Data of a Target

- Rules are used to describe targets of a given type, like a C++ library
- Targets are given by

Rules: Data of a Target

- Rules are used to describe targets of a given type, like a C++ library
- Targets are given by
  - the actual artifact, like `libfoo.a`

Rules: Data of a Target

- Rules are used to describe targets of a given type, like a C++ library
- Targets are given by
  - the actual artifact, like `libfoo.a`
  - additional files that should be installed with the target, like headers

Rules: Data of a Target

- Rules are used to describe targets of a given type, like a C++ library
- Targets are given by
  - the actual artifact, like `libfoo.a`
  - additional files that should be installed with the target, like headers
  - any additional information needed to use the target
    (no reflection on the dependency graph!)
    - Headers of public dependencies
    - Information on how to link, including libraries depended upon
    - …

Rules: Data of a Target (Example)

```
$ cat TARGETS
{ "helloworld":
  { "type": ["@", "rules", "CC", "binary"]
  , "name": ["helloworld"]
  , "srcs": ["main.cpp"]
  , "deps": ["libhello"]
  }
, "libhello":
  { "type": ["@", "rules", "CC", "library"]
  , "name": ["hello"]
  , "srcs": ["hello.cpp"]
  , "hdrs": ["hello.hpp"]
  , "deps": ["utils"]
  }
, "utils":
  { "type": ["@", "rules", "CC", "library"]
  , "name": ["utils"]
  , "srcs": ["utils.cpp"]
  , "hdrs": ["utils.hpp"]
  }
}
$
```

Rules: Data of a Target (Example)

```
$ just analyse -C repos.json libhello
INFO: Requested target is [["@","","","libhello"],{}]
INFO: Result of target [["@","","","libhello"],{}]: {
        "artifacts": {
          "libhello.a": {"data":{"id":"0681d370b705849aabe780ed74877025be591085","path":"libhello.a"},"type":"ACTION"}
        },
        "provides": {
          "compile-deps": {
            "utils.hpp": {"data":{"path":"utils.hpp","repository":""},"type":"LOCAL"}
          },
          "link-args": [
            "libhello.a",
            "libutils.a"
          ],
          "link-deps": {
            "libutils.a": {"data":{"id":"58c3d737f349042c3a5c4af2c2f3ca1d461e883e","path":"libutils.a"},"type":"ACTION"}
          }
        },
        "runfiles": {
          "hello.hpp": {"data":{"path":"hello.hpp","repository":""},"type":"LOCAL"}
        }
      }
$
```

Rule Language

Rules are mainly given by a functional expression defining this value; language

Rule Language

Rules are mainly given by a functional expression defining this value; language

- variables, `let*`-binding, conditional expressions, ...
- constructor functions for lists, maps, ...
- standard operations: accessor functions, concatenation, iteration (lists/maps), `foldl`, (conflict-free) map union, `nub_right`, ...

Rule Language

Rules are mainly given by a functional expression defining this value; language

- variables, `let*`-binding, conditional expressions, ...
- constructor functions for lists, maps, ...
- standard operations: accessor functions, concatenation, iteration (lists/maps), `foldl`, (conflict-free) map union, `nub_right`, ...
- Accessor functions to the data of the targets in the target fields

Rule Language

Rules are mainly given by a functional expression defining this value; language

- variables, `let*`-binding, conditional expressions, ...
- constructor functions for lists, maps, ...
- standard operations: accessor functions, concatenation, iteration (lists/maps), `foldl`, (conflict-free) map union, `nub_right`, ...
- Accessor functions to the data of the targets in the target fields
- actions are a means to define artifacts

Rule Language

Rules are mainly given by a functional expression defining this value; language

- variables, `let*`-binding, conditional expressions, ...
- constructor functions for lists, maps, ...
- standard operations: accessor functions, concatenation, iteration (lists/maps), `foldl`, (conflict-free) map union, `nub_right`, ...
- Accessor functions to the data of the targets in the target fields
- actions are a means to define artifacts
  - function returning a map of the output artifacts

Rule Language

Rules are mainly given by a functional expression defining this value; language

- variables, `let*`-binding, conditional expressions, ...
- constructor functions for lists, maps, ...
- standard operations: accessor functions, concatenation, iteration (lists/maps), `foldl`, (conflict-free) map union, `nub_right`, ...
- Accessor functions to the data of the targets in the target fields
- actions are a means to define artifacts
  - function returning a map of the output artifacts
  - inputs: stage of input artifacts, command vector, environment, expected outputs

Rule Language

Rules are mainly given by a functional expression defining this value; language

- variables, `let*`-binding, conditional expressions, ...
- constructor functions for lists, maps, ...
- standard operations: accessor functions, concatenation, iteration (lists/maps), `foldl`, (conflict-free) map union, `nub_right`, ...
- Accessor functions to the data of the targets in the target fields
- actions are a means to define artifacts
  - function returning a map of the output artifacts
  - inputs: stage of input artifacts, command vector, environment, expected outputs
  - mathematical function $\rightsquigarrow$ intensional equality on artifacts

Equality: Intensional versus Extensional

```
$ cat TARGETS
{ "foo":
  { "type": "generic"
  , "outs": ["out.txt"]
  , "cmds": ["echo Hello World > out.txt"]
  }
, "bar":
  { "type": "generic"
  , "outs": ["out.txt"]
  , "cmds": ["echo Hello World > out.txt"]
  }
, "baz":
  { "type": "generic"
  , "outs": ["out.txt"]
  , "cmds": ["echo -n Hello > out.txt && echo ' World' >> out.txt"]
  }
, "foo upper":
  { "type": "generic"
  , "deps": ["foo"]
  , "outs": ["upper.txt"]
  , "cmds": ["cat out.txt | tr a-z A-Z > upper.txt"]
  }
, "bar upper":
  { "type": "generic"
  , "deps": ["bar"]
  , "outs": ["upper.txt"]
  , "cmds": ["cat out.txt | tr a-z A-Z > upper.txt"]
  }
, "baz upper":
  { "type": "generic"
  , "deps": ["baz"]
  , "outs": ["upper.txt"]
  , "cmds": ["cat out.txt | tr a-z A-Z > upper.txt"]
  }
, "ALL":
  { "type": "install"
  , "files":
    {"foo.txt": "foo upper", "bar.txt": "bar upper", "baz.txt": "baz upper"}
  }
}
$
```

Equality: Intensional versus Extensional

```
$ cat TARGETS && just build -J 1
{ "foo":
  { "type": "generic"
  , "outs": ["out.txt"]
  , "cmds": ["echo Hello World > out.txt"]
  }
, "bar":
  { "type": "generic"
  , "outs": ["out.txt"]
  , "cmds": ["echo Hello World > out.txt"]
  }
, "baz":
  { "type": "generic"
  , "outs": ["out.txt"]
  , "cmds": ["echo -n Hello > out.txt && echo ' World' >> out.txt"]
  }
, "foo upper":
  { "type": "generic"
  , "deps": ["foo"]
  , "outs": ["upper.txt"]
  , "cmds": ["cat out.txt | tr a-z A-Z > upper.txt"]
  }
, "bar upper":
  { "type": "generic"
  , "deps": ["bar"]
  , "outs": ["upper.txt"]
  , "cmds": ["cat out.txt | tr a-z A-Z > upper.txt"]
  }
, "baz upper":
  { "type": "generic"
  , "deps": ["baz"]
  , "outs": ["upper.txt"]
  , "cmds": ["cat out.txt | tr a-z A-Z > upper.txt"]
  }
, "ALL":
  { "type": "install"
  , "files":
    {"foo.txt": "foo upper", "bar.txt": "bar upper", "baz.txt": "baz upper"}
  }
}
INFO: Requested target is [["@","","","ALL"],{}]
INFO: Analysed target [["@","","","ALL"],{}]
INFO: Export targets found: 0 cached, 0 uncached, 0 not eligible for caching
INFO: Discovered 4 actions, 0 trees, 0 blobs
INFO: Building [["@","","","ALL"],{}].
INFO: Processed 4 actions, 1 cache hits.
INFO: Artifacts built, logical paths are:
        bar.txt [4e3dffe834ac70600a7cb71fbc1f6a694c9d041f:12:f]
        baz.txt [4e3dffe834ac70600a7cb71fbc1f6a694c9d041f:12:f]
        foo.txt [4e3dffe834ac70600a7cb71fbc1f6a694c9d041f:12:f]
$
```

Rules: More on Actions

- Consider actions as functions: better check they are!
  ⇝ `just rebuild`: rebuild everything, comparing against cache
  - possibly against a different remote-exeuction endpoint
  - possibly with a different local-launcher prefix
    (e.g., `["env", "LDPRELOAD=...", ..., "--"]` to use `libfaketime`, `disorderfs`, ...)

Rules: More on Actions

- Consider actions as functions: better check they are!
  ⇝ `just rebuild`: rebuild everything, comparing against cache
  - possibly against a different remote-exeuction endpoint
  - possibly with a different local-launcher prefix
    (e.g., `["env", "LDPRELOAD=...", ..., "--"]` to use `libfaketime`, `disorderfs`, ...)
- Allow special non-pure “tainted” actions
  ... but require target/rules to declare (transitive) taintedness
  - accept (but report) failure, provided required outputs are present,
    but only cache on success, e.g., test actions (“we build the test report”)
  - never cache, e.g., monitoring actions, check tests for flakyness

Anonymous Targets

- Interface API generation (think protobuf)
  - abstract description of wire format, possibly depending on other descriptions
  - can generate APIs for various languages

Anonymous Targets

- Interface API generation (think protobuf)
  - abstract description of wire format, possibly depending on other descriptions
  - can generate APIs for various languages

? What is the value of such an interface target?
Should not have to know the languages that will use that format later!

Anonymous Targets

- Interface API generation (think `protobuf`)
  - abstract description of wire format,
    possibly depending on other descriptions
  - can generate APIs for various languages

? What is the value of such an interface target?
Should not have to know the languages that will use that format later!

⇝ Take the dependency graph with just the files (and abstract rule labels) as value
… generate actual targets by binding rule labels to actual rules

- targets are not associated with a specific location anyway
- equality: intensional equality of node and locational equality of rules
  → no duplication if binding for the same language requested several times.

Anonymous Targets

```
, "library":
  { "doc": ["A C++ libaray"]
  , "target_fields": ["srcs", "hdrs", "private-hdrs", "deps", "proto"]
  , /* ... */
  , "anonymous":
    { "proto-deps":
      { "target": "proto"
      , "provider": "proto"
      , "rule_map":
        { "library": ["./", "proto", "library"]
        , "service library": ["./", "proto", "service library"]
        }
      }
    }
  , /* ... */
  }
```

- Interface API generation (think protobuf)
  - abstract description of wire format, possibly depending on other descriptions
  - can generate APIs for various languages

? What is the value of such an interface target?
Should not have to know the languages that will use that format later!

⇝ Take the dependency graph with just the files (and abstract rule labels) as value
… generate actual targets by binding rule labels to actual rules

- targets are not associated with a specific location anyway
- equality: intensional equality of node and locational equality of rules
  → no duplication if binding for the same language requested several times.

Sources

- https://github.com/just-buildsystem/justbuild
- License: Apache 2.0