Using buck to build Rust projects

Apr 13 2023

A few days ago, Facebook/Meta/idk announced that buck2 is now open source (https://engineering.fb.com/2023/04/06/open-source/buck2-open-source-large-scale-build-system/).

> *"Buck2 is an extensible and performant build system written in Rust and designed to make your build experience faster and more efficient."*

As it just so happens, I have taken an increasing interest in build systems lately. I have mostly ignored the cool kids' build system things, because I have never worked at a FAANG, or at a true monorepo shop. I also personally try and avoid the JVM wherever possible, and the first generation of these tools were all built on top of it. (Yes, that bias may be outdated, I'm not saying *you* should avoid the JVM, just stating my own bias up front.)

So this timing was perfect! Let's explore what using buck looks like.

> *"A brief aside: listen, I have no love for Facebook. I deleted my account (im-deleting-my-facebook-tonight) almost a dozen years ago. That doesn't mean I'm not interested in using good tools they produce. If you feel differently, fine, but that's not what I want to talk about today, so I'm not going to."*

Oh, one last bit before I begin: I'm not going to do a lot of motivating on "why would you want to use Buck?" in this post. There's a few reasons for that, but for now, if that's what you're looking for, this post isn't it. We're doing this purely for the novelty of trying out some new tech right now. I will probably end up with a post giving better motivations at some point in the future, but I think it makes more sense once you see how it works, rather than starting there.

A series

This post is part of a series:

- Using buck to build Rust projects you are here
- Using Crates.io with Buck (using-cratesio-with-buck)
- Updating Buck (updating-buck)

This post represents how to do this at the time that this was posted; future posts may update or change something that happens here. Here's a hopefully complete but possibly incomplete list of updates and the posts that talk about it:

- `buck2 init` also creates a file named `.buckroot`, see "Updating Buck"

Getting started with buck2

The Getting Started page (https://buck2.build/docs/getting_started/) will give you instructions on installing buck. As of this moment, the instructions are:

```
$ rustup install nightly-2023-01-24
$ cargo +nightly-2023-01-24 install --git https://github.com/
```

This is mega-convenient for me as a Rust user, but probably not if you don't have Rust installed. That said, this is a first release, and so I don't expect anything fancier. This is what `cargo install` is good for!

Let’s make a new directory, **`hello`**:

```
$ mkdir buck-rust-hello
$ cd buck-rust-hello
```

To initialize a project, we use this command:

```
$ buck2 init --git
```

Before we move forward, let’s examine what this generated for us.

Initial project files

We now have this stuff in our directory:

```
$ git add .
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .buckconfig
        new file:   .gitignore
        new file:   .gitmodules
        new file:   BUCK
        new file:   prelude
        new file:   toolchains/BUCK
```

Let’s talk about each of these in turn.

.buckconfig

The `.buckconfig` file is… a configuration file for Buck, go figure. It looks like this:

```
[repositories]
root = .
prelude = prelude
toolchains = toolchains
none = none

[repository_aliases]
config = prelude
fbcode = none
fbsource = none
buck = none

[parser]
target_platform_detector_spec = target:root//...->prelude//pl
```

That `none = none` is kind of amusing. Regardless of that, this file is extremely important: it configures the entire thing. In a sense, it's like `Cargo.toml`: just like a package is defined by the existence of a `Cargo.toml`, a `.buckconfig` defines the existence of a… cell. Which defines a package. We'll get there. Point is, this is the top level configuration. We say that the repository root is in the current directory, we'd like to use the default prelude and toolchains.

I uh… I don't know what the `none = none` is for. It might just be a bug. I haven't seen it in some of the other config files I've poked at. Let's just leave that alone for now. I do have a suspicion though… and it involves the next section.

We also have a table for repository aliases. I couldn't find any documentation on this, but I would imagine this means we could use the name `config` instead of `prelude` later. Looks like we don't have any way to refer to `fbcode` and `fbsource`, which makes sense, and same with `buck`.

(I wonder if this is what the `none = none` is about above, defining a sort of "none repository" that we can then alias these to.)

Finally, we have a parser table, with one entry, pointing out where a thing exists. I know this configures Buck's parser, but other than that… I'm sure I'll figure it out eventually.

`.gitmodules` & `prelude`

We have a git submodule, pointing to `https://github.com/facebook/buck2-prelude.git`, that lives at the `prelude` directory. If you poke around in there, you'll find a bunch of `.bzl` files that implement useful features for us to use. We'll get into those in a moment, but the point is that this is sort of like a 'standard library' if you will. You could also not use it and define your own. If you're that kind of person.

`.gitignore`

A very simple `.gitignore` will be created, that contains one line: `/buck-out`. This is where buck stores artifacts produced by your builds, so we don't want that checked into version control.

`BUCK`

Now we get to the good stuff. Here's the generated `BUCK` file:

```
# A list of available rules and their signatures can be found

genrule(
    name = "hello_world",
    out = "out.txt",
    cmd = "echo BUILT BY BUCK2> $OUT",
)
```

`genrule` is like a function, provided by our prelude. If you're curious, the implementation is in `prelude/genrule.bzl`. This command, as you may imagine, sets up a rule, named `hello_world`, that produces a file called `out.txt`. It does this by running the `cmd`. Nice and straightforward. We'll give that a try in a moment, but first, one last file:

`toolchains/BUCK`

This file describes a toolchain. Here's the contents:

```
load("@prelude//toolchains:genrule.bzl", "system_genrule_tool

system_genrule_toolchain(
    name = "genrule",
    visibility = ["PUBLIC"],
)
```

This loads a certain rule from the prelude, and then defines this as a public toolchain. We can define as many toolchains as we want here, for example, if we wanted to build both Rust and Python, we could define both toolchains here for later use.

The "genrule" toolchain is used to generate files from a shell command, as we saw before with our rule that produces `out.txt`. So, in my understanding, here we are defining that we wish to actually use that. And then, in the `BUCK` file, we're using this toolchain to implement our rule.

Invoking our first build

Okay, let's actually give this a shot. To instruct Buck to build something, we invoke it with the "target pattern" as an argument. Let's ask Buck what targets it knows how to build. To do this:

```
C:\Users\steve\Documents\GitHub\buck-rust-hello〉 buck2 target
Build ID: cd778a29-2ba4-484b-8956-dc67f6fc0625
Jobs completed: 4. Time elapsed: 0.0s.
root//:hello_world
```

The `//...` is a "target pattern." The `/...` means "all build targets in build files in subdirectories", and `/` means our root directory, so `//...` means "all targets in all build files in all subdirectories." By passing this target to `buck2 targets`, we can see every target in the project. This shows our one target we've defined, `root://:hello_world`. This name was defined in our `BUCK` file above. If you change that to

```
genrule(
    name = "lol",
```

then `buck2 targets //...` would show `root://:lol`.

Let's actually build our target:

```
〉 buck2 build //:hello_world
File changed: root//BUCK
Build ID: 73f4b797-2238-47bc-8e43-7ffcb2b7d9b7
Jobs completed: 36. Time elapsed: 0.0s. Cache hits: 0%. Comma
BUILD SUCCEEDED
```

Okay the build succeeded, but where is our `out.txt`? We can ask buck!

```
〉 buck2 build //:hello_world --show-output
Build ID: 7ce93845-ab1e-4b0a-9274-51fed9f9e295
Jobs completed: 3. Time elapsed: 0.0s.
BUILD SUCCEEDED
root//:hello_world buck-out\v2\gen\root\fb50fd37ce946800\__he
```

It lives in a deeply nested subdirectory of `buck-out`, a new top-level directory that was created for us. If you remember from before, this directory is ignored in our `.gitignore`.

If we look at the file, you can see it contains the text we wanted it to contain.

Let's build a second time!

```
〉 buck2 build //:hello_world
File changed: root//.git/index.lock
File changed: root//.git
File changed: root//.git/modules/prelude/index.lock
31 additional file change events
Build ID: c00e4bfa-a1f8-40c7-a61c-2a394dca5da5
Jobs completed: 5. Time elapsed: 0.0s.
BUILD SUCCEEDED
```

Buck has noticed that we've changed some files, but since our rule doesn't depend on any of them, we're good to go.

Building some Rust code

Okay, `echo` to a file is fun, but let's actually build some Rust. Create a file, `hello.rs`:

```
fn main() {
    println!("Hello, world!");
}
```

and then update the `BUCK` file to this:

```
rust_binary(
    name = "hello_world",
    srcs = ["hello.rs"],
    crate_root = "hello.rs",
)
```

This says “hey, we’re building a Rust binary, it has this target name, these source files, and the crate root lives here.” Given we only have one file, there’s some reptition. It happens. Let’s build:

```
〉buck2 build //:hello_world
File changed: root//BUCK
Error running analysis for `root//:hello_world (prelude//plat

Caused by:
    0: Error looking up configured node root//:hello_world (p
    1: Error looking up configured node toolchains//:cxx (pre
    2: looking up unconfigured target node `toolchains//:cxx`
    3: Unknown target `cxx` from package `toolchains//`.
       Did you mean one of the 1 targets in toolchains//:BUCK
Build ID: f126ce07-efe8-41d3-8aae-8b7d31a4dafc
Jobs completed: 4. Time elapsed: 0.0s.
BUILD FAILED
```

Oops! We didn’t set up a rust toolchain! Let’s do that now. Edit `toolchains/BUCK`:

```
load("@prelude//toolchains:rust.bzl", "system_rust_toolchain"
system_rust_toolchain(
    name = "rust",
    default_edition = "2021",
    visibility = ["PUBLIC"],
)
```

And… when we build again, the same error. Now. I am not 100% sure what's going on here, but this is what we need to do:

```
load("@prelude//toolchains:rust.bzl", "system_rust_toolchain"
load("@prelude//toolchains:genrule.bzl", "system_genrule_tool
load("@prelude//toolchains:cxx.bzl", "system_cxx_toolchain")
load("@prelude//toolchains:python.bzl", "system_python_bootst

system_genrule_toolchain(
    name = "genrule",
    visibility = ["PUBLIC"],
)

system_rust_toolchain(
    name = "rust",
    default_edition = "2021",
    visibility = ["PUBLIC"],
)

system_cxx_toolchain(
    name = "cxx",
    visibility = ["PUBLIC"],
)

system_python_bootstrap_toolchain(
    name = "python_bootstrap",
    visibility = ["PUBLIC"],
)
```

I *believe* that this is because, to compile the Rust compiler, we need Python and a C++ compiler. Well, I *did* believe that, but after digging into things some more, it's that the Rust toolchain from the prelude depends on the CXX toolchain in the prelude, because the Rust toolchain invokes the C compiler to invoke the linker. I'm still not 100% sure why Python needs to be in there. Anyway.

Now, when I run, I got this:

```
〉 buck2 build //:hello_world -v 3
Action failed: prelude//python_bootstrap/tools:win_python_wra
Internal error: symlink(original=../../../../../../../../../p
Build ID: 57a66885-f7e7-474b-a782-b49fc4425be9
Jobs completed: 14. Time elapsed: 0.0s.
BUILD FAILED
Failed to build 'prelude//python_bootstrap/tools:win_python_w
```

I got this becuase I'm on Windows, and Windows restricts the ability to create symlinks by default. Turning on "Developer Mode" (which I'm surprised that I haven't had to turn on so far yet), I get further:

```
<whole bunch of output>
  = note: 'clang++' is not recognized as an internal or exter
          operable program or batch file.
```

Here's that "invokes the compiler to get the linker" thing I was referring to above.

Now... by default, the Rust support is for the GNU version of the Windows toolchain. I never use that. Upstream has said that they want everything to be supported, so that change will come at some point, maybe by the time you read this! But in the meantime, I could get my (pure Rust) projects building with two small patches:

```
diff --git a/prelude/toolchains/cxx.bzl b/prelude/toolchains/
index c57b7b8..dc14ca7 100644
--- a/prelude/toolchains/cxx.bzl
+++ b/prelude/toolchains/cxx.bzl
@@ -39,7 +39,7 @@ def _system_cxx_toolchain_impl(ctx):
         CxxToolchainInfo(
             mk_comp_db = ctx.attrs.make_comp_db,
             linker_info = LinkerInfo(
-                linker = RunInfo(args = ["clang++"]),
+                linker = RunInfo(args = ["link"]),
                 linker_flags = ["-fuse-ld=lld"] + ctx.attrs.
                 archiver = RunInfo(args = ["ar", "rcs"]),
                 archiver_type = archiver_type,
diff --git a/prelude/toolchains/rust.bzl b/prelude/toolchains
index 8172090..4545d55 100644
--- a/prelude/toolchains/rust.bzl
+++ b/prelude/toolchains/rust.bzl
@@ -23,7 +23,7 @@ _DEFAULT_TRIPLE = select({
         # default when we're able; but for now buck2 doesn't
         # toolchain yet.
         "config//cpu:arm64": "aarch64-pc-windows-gnu",
-        "config//cpu:x86_64": "x86_64-pc-windows-gnu",
+        "config//cpu:x86_64": "x86_64-pc-windows-msvc",
     }),
 })
```

Now a build works!

```
〉 buck2 build //:hello_world
File changed: root//BUCK
File changed: root//.git/index.lock
File changed: root//.git
6 additional file change events
Build ID: 65fc80aa-7bfa-433a-bfa7-57919147b550
Jobs completed: 65. Time elapsed: 1.0s. Cache hits: 0%. Comma
BUILD SUCCEEDED
```

We can run it to see the output:

```
〉 buck2 run //:hello_world
Build ID: 78b0ca23-2c7c-4c02-a161-bba15e3b38bd
Jobs completed: 3. Time elapsed: 0.0s.
hello world
```

Same idea as `cargo run`.

Speaking of `cargo run`, what might this look like with Cargo? Well, we can create a `Cargo.toml`:

```
[package]
name = "hello_world"
version = "0.1.0"
edition = "2021"

[[bin]]
name = "hello_world"
path = "hello.rs"
```

and try it out. Oh, and you'll probably want to put `target` into your `.gitignore`.

Let's build. The "benchmark" command in nushell is sort of like `time` on a UNIX system:

```
〉benchmark { cargo build }
   Compiling hello_world v0.1.0 (C:\Users\steve\Documents\Git
    Finished dev [unoptimized + debuginfo] target(s) in 0.34s
416ms 490us 100ns
〉benchmark { cargo build }
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
77ms 317us 200ns
```

Not too bad, a bit under half a second for the initial build, and near immediate on a subsequent build. What about buck?

```
〉benchmark { buck2 build //:hello_world -v 3 }
Running action: <snip>
Build ID: 47ebd9f1-3394-4f72-a0fb-02c936035d2b
Jobs completed: 58. Time elapsed: 0.8s. Cache hits: 0%. Comma
BUILD SUCCEEDED
945ms 557us
〉benchmark { buck2 build //:hello_world -v 3 }
Build ID: 5eed24e8-a540-454f-861a-855464aba3c9
Jobs completed: 3. Time elapsed: 0.0s.
BUILD SUCCEEDED
78ms 274us 100ns
```

Not too shabby; Buck is a *teeny* bit slower on the initial build, but when stuff is cached, both are the same speed. All of this is fast enough to qualify as "basically instant."

Adding a library

Let's up the complexity a bit, by adding a library that we want to depend on. Here's a **`lib.rs`**:

```
pub fn print_hello() {
    println!("Hello, world!");
}
```

We want to change our top-level **`BUCK`** to add this:

```
+rust_library(
+    name = "print_hello",
+    srcs = ["lib.rs"],
+    edition = "2021",
+    visibility = ["PUBLIC"],
+)
+
 rust_binary(
     name = "hello_world",
     srcs = ["hello.rs"],
     crate_root = "hello.rs",
+    deps = [
+        ":print_hello",
+    ],
 )
```

Here, we make a new library, **`print_hello`**, and then make our binary depend on it.

Let's change the code in **`main.rs`** to use the library:

```
fn main() {
    println!("hello world");
    print_hello::print_hello();
}
```

And that's it! Let's examine our targets:

```
❯ buck2 targets //...
Build ID: 4646f2e7-0ea3-4d59-8590-3da0708ce96e
Jobs completed: 4. Time elapsed: 0.0s.
root//:hello_world
root//:print_hello
```

They're both there! We can now build one, the other, or everything:

```
# build everything
❯ buck2 build //...
# build just the library
❯ buck2 build //:print_hello
# build 'just' the binary, this will of course end up buildin
❯ buck2 build //:print_hello
```

Let's make sure it still prints our output:

```
❯ buck2 run //:hello_world
Build ID: d76c80fb-dd77-463a-86a1-b6443cea10f6
Jobs completed: 3. Time elapsed: 0.0s.
Hello, world!
```

Nice.

Let's compare that to Cargo. Modify `Cargo.toml`:

```
[lib]
name = "print_hello"
path = "lib.rs"
```

and build:

```
〉 cargo run
   Compiling hello_world v0.1.0 (C:\Users\steve\Documents\Git
    Finished dev [unoptimized + debuginfo] target(s) in 0.42s
     Running `target\debug\hello_world.exe`
Hello, world!
```

Nice.

more cargo-ish

Before we move forward, this isn't *exactly* an apples to apples comparison; we've been doing a lot of configuration for Cargo that we normally wouldn't have to do, and also, what if you already have a Cargo project, but you want to try out Buck with it?

Do this:

```
> mkdir src
> mv hello.rs src/main.rs
> mv lib.rs src/main.rs
```

And delete the configuration from `Cargo.toml`, leaving just the `package` table. Finally, we need to change `src/main.rs`, given that we're using the default crate name for the library crate, which is `hello_world` and not `print_hello`:

```rust
fn main() {
    hello_world::print_hello();
}
```

After this, `cargo build` works just fine. But what about Buck?

So. We have a bit of weirdness here, and I'm not sure if it's actually work-around-able in Buck or not, since I'm still learning this myself. But if we do the basic translation, we'll get an error. Let's try it. This is how you modify the `BUCK` file:

```
rust_library(
    name = "hello_world",
    srcs = glob(["src/**/*.rs"]),
    edition = "2021",
    visibility = ["PUBLIC"],
)

rust_binary(
    name = "hello_world",
    srcs = ["src/main.rs"],
    crate_root = "src/main.rs",
    deps = [
        ":print_hello",
    ],
)
```

Cargo produces a binary and a library, both called `hello_world`, but buck doesn't like that:

```
❯ buck2 run //:hello_world
Error evaluating build file: `root//:BUCK`

Caused by:
    Traceback (most recent call last):
      * BUCK:8, in <module>
          rust_binary(
    error: Attempted to register target root//:hello_world tw
      --> BUCK:8:1
```

```
     |
   8 | / rust_binary(
   9 | |     name = "hello_world",
  10 | |     srcs = ["src/main.rs"],
  11 | |     crate_root = "src/main.rs",
  12 | |     deps = [
  13 | |         ":print_hello",
  14 | |     ],
  15 | | )
     | |_^
     |

Build ID: d6a8925d-0180-4308-bcb9-fbc888e7eca1
Jobs completed: 4. Time elapsed: 0.0s.
BUILD FAILED
```

It's like hey! You have two targets named `hello_world`! That's confusing! It also reveals a difference between Buck and Cargo. With Cargo, if you remember our configuration, we had to point it to the crate root. From there, Cargo just leans on `rustc` to load up all of the other files that may be required if you have a bunch of modules. But with Buck, we need to tell it up front which files we use. So as you can see above:

```
    srcs = glob(["src/**/*.rs"]),
```

We can use the `glob` command to glob up all of our files, which is nice, but it's… it's actually wrong. We want to glob everything *except* `main.rs`. If `main.rs` were to change, this would try and re-build both the binary and the library, in my understanding. So that's annoying.

It's not just annoying for Buck, though. Having both a `src/main.rs` and a `src/lib.rs` has led to so much confusion from beginners over the years. At some point, someone puts `mod lib;` into `src/main.rs` and everything goes to hell. The original intention of this layout, to make simple things simple, is a good idea, but I think that sadly, we missed the mark here. Luckily, Cargo also supports a `bin` directory:

```
> mkdir src/bin
> mv src/main.rs src/bin
```

We can still `cargo run` and `cargo build` and this all works just fine. This doesn't fix our glob issue, though, because `src/bin` is still inside of `src`.

I *think*, in the abstract, I'd prefer a layout like `src/{lib,bin}`. You want things to not really be nested. So let's do that. Both Cargo and Buck can handle it! It's just not as nice as being purely default in Cargo, since that convention is so strong.

```
# if you didn't do this above
> mkdir src/bin
> mv src/main.rs src/bin

> mkdir src/lib
> mv src/lib.rs src/lib/print_hello.rs
```

We have to change `src/bin/main.rs` to use `hello_world` again

```
fn main() {
    hello_world::print_hello();
}
```

And we have to re-add some configuration into `Cargo.toml`:

```
[lib]
```

```
path = "src/lib/print_hello.rs"
```

Everything should build just fine. But what about Buck?

So, once you start getting into subdirectories, you can also start using multiple `BUCK` files. So we can empty out our root `BUCK` file (I'm leaving it existing but empty, if you want to delete it you can but you'll recreate it in the next part anyway), and create two new ones. Here's `src/lib/BUCK`:

```
rust_library(
    name = "hello_world",
    srcs = glob(["**/*.rs"]),
    crate_root = "print_hello.rs",
    edition = "2021",
    visibility = ["PUBLIC"],
)
```

and `src/bin/BUCK`:

```
rust_binary(
    name = "hello_world",
    srcs = ["main.rs"],
    crate_root = "main.rs",
    deps = [
        ":hello_world",
    ],
)
```

We added in a `crate_root` to the library as well. Okay, let's try this:

```
❯ buck2 run //:hello_world
File changed: root//src/lib/BUCK
```

```
Unknown target `hello_world` from package `root//`.
Did you mean one of the 0 targets in root//:BUCK?
Build ID: d5059fc9-8001-47c4-ba5a-6ba605a4182c
Jobs completed: 2. Time elapsed: 0.0s.
BUILD FAILED
```

Oops! Since we moved files around, the names of our targets have changed. Let's examine them:

```
❯ buck2 targets //...
Build ID: c4165964-cb87-49b4-8afe-4a3fc2c526bc
Jobs completed: 4. Time elapsed: 0.0s.
root//src/bin:hello_world
root//src/lib:hello_world
```

We had only seen very basic target patterns, but this is enough to show off:

`root/src/bin:hello_world`

can be read as

> *"The "hello_world" target defined in `/src/bin/BUCK`."*

Our target names changing also means we made one mistake in our new `BUCK` files. Let's give it a try:

```
❯ buck2 run //src/bin:hello_world
Error running analysis for `root//src/bin:hello_world (prelud

Caused by:
    0: Error looking up configured node root//src/bin:hello_w
    1: Cyclic computation detected when computing key `root//
Build ID: 930ab541-c2dd-44f5-aef1-f6658a2b7c53
```

```
Jobs completed: 2. Time elapsed: 0.0s.
BUILD FAILED
```

Right. Our binary depends on `:hello_world`, which it is itself named `hello_world`, so there's a problem. But that's just it, we don't want to depend on any old `hello_world`, we want to depend on our libary. Can you write out the target pattern that should go in `src/bin/BUCK`?

It looks like this:

```
    deps = [
        "//src/lib:hello_world",
    ],
```

"The `hello_world` target in `/src/lib/BUCK`. Perfect. And now it works!

```
〉buck2 run //src/bin:hello_world
File changed: root//src/bin/BUCK
Build ID: c6d2fdaa-298a-425a-9091-d3f6b38c4336
Jobs completed: 12. Time elapsed: 0.5s. Cache hits: 0%. Comma
Hello, world!
```

It kinda stinks to have to type all of that out. Luckily, Buck supports aliases for target patterns. Take our top-level `BUCK` file, and add this:

```
alias(
    name = "build",
    actual = "//src/bin:hello_world",
    visibility = ["PUBLIC"],
)
```

And now we can use it:

```
❯ buck2 build
Build ID: a87ed1e2-cfab-47b0-830e-407217997fd7
Jobs completed: 2. Time elapsed: 0.0s.
BUILD SUCCEEDED
```

Fun.

Okay! This is getting a bit long, so let’s end there. We have more to learn before buck can actually replace Cargo in our normal development workflow, but I hope this helped you see how you could get started with Buck if you wanted to.

If you want to check out this on your own, I’ve published this on GitHub: https://github.com/steveklabnik/buck-rust-hello/tree/024ef54ba45627e87a65aaf2f69c6661198c336c (https://github.com/steveklabnik/buck-rust-hello/tree/024ef54ba45627e87a65aaf2f69c6661198c336c)

Next up, we’ll be tackling other features, like “using crates from crates.io.” No promises on when that’ll get published though!