Karol Kuczmarski

fn(Tea) -> Code

Add examples to your Rust libraries

Posted on Wed 28 February 2018 in Code

When you're writing a library for other programs to depend on, it is paramount to think how the developers are going to use it in their code.

The best way to ensure they have a pleasant experience is to *put yourself in their shoes*. Forget the internal details of your package, and consider only its outward interface. Then, come up with a realistic use case and just *implement it*.

In other words, you should create complete, end-to-end, and (somewhat) usable *example applications*.

Examples are trouble

You may think this is asking a lot, and I wouldn't really disagree here.

In most languages and programming platforms, it is indeed quite cumbersome to create example apps. This happens for at least several different reasons:

- **It typically requires bootstrapping an entire project from scratch.** If you are lucky, you will have something like `create-react-app` to get you going

relatively quickly. Still, you need to wire up the new project so that it depends on the *source code* of your library rather than its published version, and this tends to be a non-standard option — if it is available at all.

- **It's unclear where should the example code live.** Should you just throw it away, once it has served its immediate purpose? I'm sure this would discourage many people from creating examples in the first place. It's certainly better to keep them in the version control, allowing their code to serve as additional documentation.

  But if you intend to do this, you need to be careful not to deploy the example along with your library when you upload it to the package registry for your language. This may require maintaining an explicit blacklist and/or whitelist, in the vein of `MANIFEST` files in Python.

- **Examples may break as the library changes.** Although example apps aren't integration tests that have a clear, expected outcome, they should at the very least *compile correctly*.

  The only way to ensure that is to include them in the build/test pipeline of your library. To accomplish this, however, you may need to complicate your CI setup, perhaps by introducing additional languages like Bash or Python.

- **It's harder to maintain quality of example code.** Any linters and static analyzers that you're normally running will likely need to be configured to also apply to the examples. On the other hand, however, you probably don't want those checkers to be *too strict* (it's just example code, after all), so you may want to turn off some of the warnings, adjust the level of others, and so on.

So essentially, writing examples involves quite a lot of hassle. It would be great if the default tooling of your language helped to lessen the burden at least a *little* bit.

Well, good news! If you're a Rust programmer, the language has basically got you covered.

Cargo — the standard build tool and package manager for Rust — has some dedicated features to support *examples* as a first-class concept. While it doesn't completely address all the pain points outlined above, it goes a long way towards minimizing them.

What are Cargo examples?

In Cargo's parlance, an *example* is nothing else but a Rust source code of a standalone executable[1] that typically resides in a single `.rs` file. All such files should be places in the `examples/` directory, at the same level as `src/` and the `Cargo.toml` manifest itself[2].

Here's the simplest example of, ahem, an *example*:

```
// examples/hello.rs
fn main() {
    println!("Hello from an example!");
}
```

You can run it through the typical `cargo run` command; simply pass the example name after the `--example` flag:

```
$ cargo run --example hello
Hello from an example!
```

It is also possible to run the example with some additional arguments:

```
$ cargo run --example hello2 -- Alice
Hello, Alice!
```

which are relayed directly to the underlying binary:

```
// examples/hello2.rs
use std::env;

fn main() {
    let name = env::args().skip(1).next();
    println!("Hello, {}!", name.unwrap_or("world".into()));
}
```

As you can see, the way we run examples is very similar to how we'd run the `src/bin` binaries, which some people use as normal entry points to their Rust programs.

The important thing is that you don't have to worry what to do with your example code anymore. All you need to do is drop it in the `examples/` directory, and let Cargo do the rest.

Dependency included

Of course in reality, your examples will be at least a *little* more complicated than that. For one, they will surely call into your library to use its API, which means they

need to depend on it & import its symbols.

Fortunately, this doesn't complicate things even one bit.

The library crate itself is already an *implied dependency* of any code inside the `examples/` directory. This is automatically handled by Cargo, so you don't have to modify `Cargo.toml` (or do anything else really) to make it happen.

So without any additional effort, you can just to link to your library crate in the usual manner, i.e. by putting `extern crate` on top of the Rust file:

```
// examples/real.rs
extern crate mylib;

fn main() {
    let thing = mylib::make_a_thing();
    println!("I made a thing: {:?}", thing);
}
```

This goes even further, and extends to any dependency *of the library itself*. All such third-party crates are automatically available to the example code, which proves handy in common cases such as Tokio-based asynchronous APIs:

```
// example/async.rs
extern crate mylib;
extern crate tokio_core;  // assuming it's in mylib's [dependenci

fn main() {
    let mut core = tokio_core::reactor::Core::new().unwrap();
    let thing = core.run(mylib::make_a_thing_asynchronously()).un
    println!("I made a thing: {:?}", thing);
}
```

More deps

Sometimes, however, it is very useful to pull in an additional package or two, just for the example code.

A typical case may involve *logging*.

If your library uses the usual `log` crate to output debug messages, you probably want to see them printed out when you run your examples. Since the `log` crate is just a facade, it doesn't offer any built-in way to pipe log messages to standard output. To handle this part, you need something like the `env_logger` package:

```
// example/with_logging.rs
```

```
extern crate env_logger;
extern crate mylib;

fn main() {
    env_logger::init();
    println("{:?}", mylib::make_a_thing());
}
```

To be able to import `env_logger` like this, it natually has to be declared as a dependency in our `Cargo.toml`.

We won't put it in the `[dependencies]` section of the manifest, however, as it's not needed by the library code. Instead, we should place it in a separate section called `[dev-dependencies]`:

```
[dev-dependencies]
env_logger = "0.5"
```

Packages listed there are shared by tests, benchmarks, and — yes, examples. They are not, however, linked into regular builds of your library, so you don't have to worry about bloating it with unnecessary code.

Growing bigger

So far, we have seen examples that span just a single Rust file. Practical applications tend to be bigger than that, so it'd be nice if we could provide some multi-file examples as well.

This is easily done, although for some reason it doesn't seem to be mentioned in the official docs.

In any case, the approach is identical to executables inside `src/bin/`. Basically, if we have a single `foo.rs` file with executable code, we can expand it to a `foo/` subdirectory with `foo/main.rs` as the entry point. Then, we can add whatever other submodules we want — just like we would do for a regular Rust binary crate:

```
// examples/multifile/main.rs
extern crate env_logger;
extern crate mylib;

mod util;

fn main() {
    env_logger::init();
    let ingredient = util::create_ingredient();
    let thing = mylib::make_a_thing_with(ingredient);
```

```
    println("{:?}", thing);
}
```

```
// examples/multifile/util.rs

pub fn create_ingredient() -> u64 {
    42
}
```

Of course, it won't be often that examples this large are necessary. Showing how a library can scale to bigger applications can, however, be very encouraging to potential users.

Maintaining maintainability

Thus far, we have discussed how to create small and larger examples, how to use additional third-party crates in example programs, and how to easily build & run them using built-in Cargo commands.

All this effort spent on writing examples would be of little use if we couldn't ensure that they *work*.

Like every type of code, examples are prone to breakage whenever the underlying API changes. If the library is actively developed, its interface represents a moving target. It is quite expected that changes may sometimes cause old examples to stop compiling.

Thankfully, Cargo is very dilligent in reporting such breakages. Whenever you run:

```
$ cargo test
```

all examples are built simultaneously with the execution of your regular test suite[3]. You get the compilation guarantee for your examples essentially for free — there is no need to even edit your `.travis.yml`, or to adjust your continuous integration setup in any other way!

Pretty neat, right?

This saying, you should keep in mind that simply compiling your examples on a regular basis is not a foolproof guarantee that their code never becomes outdated. Examples are *not* integration tests, and they won't catch important changes in your implementation that aren't breaking the interface.

Examples-Driven Development?

You may be wondering then, what's exactly the point of writing examples? If you got tests on one hand to verify correctness, and documentation on the other hand to inform your users, then having a bunch of dedicated executable examples may seem superfluous.

To me, however, an impeccable test suite and amazing docs — which also remain comprehensive and awesome for an entire lifetime of the library! — sound a bit too much like a perfect world :) Adding examples to the mix can almost always improve things, and their maintenance burden should, in most cases, be very minimal.

But I have also found out that starting off with examples early on is a great way to *validate the interface design*.

Once the friction of creating small test programs has been eliminated, they become indispensable for prototyping new features. Wanna try out that new thing you've just added? Simple: just make a quick example for it, run it, and see what happens!

In many ways, doing this feels similar to trying out things in a REPL — something that's almost exclusive to dynamic/interpreted languages. But unlike mucking around in Python shell, examples are *not* throwaway code: they become part of your project, and remain useful for both you & your users.

1. It is also possible to create examples which are themselves just libraries. I don't think this is particularly useful, though, since all you can do with such examples is build them, so they don't provide any additional value over normal tests (and especially doc tests). ↩
2. Because they are outside of the `src/` directory, examples do not become a part of your library's code, and are not deployed to crates.io. ↩
3. You can also run `cargo build --examples` to only compile the examples, without running any kind of tests. ↩

Rust Cargo examples documentation packaging