**Building CLI Apps in Rust — What You Should Consider**

This is part guide, part post about what you should consider before building a new CLI app in Rust.

[[](https://jondot.medium.com/?source=post_page-----99cdcc67710c--------------------------------)](https://jondot.medium.com/?source=post_page-----99cdcc67710c--------------------------------)

[[Better Programming](https://betterprogramming.pub/?source=post_page-----99cdcc67710c--------------------------------)](https://betterprogramming.pub/?source=post_page-----99cdcc67710c--------------------------------)

[Dotan Nahum](https://jondot.medium.com/?source=post_page-----99cdcc67710c--------------------------------)

·

Follow

Published in

[Better Programming](https://betterprogramming.pub/?source=post_page-----99cdcc67710c--------------------------------)

·

14 min read

·

Apr 28

382

6



Photo by [Marc Mintel](https://unsplash.com/@mmintel?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

***Like this article? have questions?***[***Follow me on twitter***](https://twitter.com/jondot)***and ask away:)***

Before jumping into a cargo new and hitting it, read up and see how to boost your development experience, CLI ergonomics, and project maintainability.

**Structure**

When you have a CLI app, you have a bunch of flags and commands, and for these the suitable logical module.

For example, for a git clone subcommand you have some *clone* functionality, but for a git commit subcommand you might have a completely different functionality that can live in a completely different module (and should). So, this can be a different module and also a different *file*.

Or, you can have a simple flat structure of a CLI app that does just one thing but takes various flags as tweaks to that one single thing:

$ demo --help  
A simple to use, efficient, and full-featured Command Line Argument Parser  
  
Usage: demo[EXE] [OPTIONS] --name <NAME>  
Options:  
 -n, --name <NAME> Name of the person to greet  
 -c, --count <COUNT> Number of times to greet [default: 1]  
 -h, --help Print help  
 -V, --version Print version  
$ demo --name Me  
Hello Me!

So when we talk about a command layout it should mean splitting to files, and then we also want to talk about file structure and project layout.

After reviewing a good number of popular CLI apps in Rust I found that there are generally three kinds of app structures:

* Ad-hoc, see [xh](https://github.com/ducaale/xh/blob/master/src/cli.rs" \t "_blank) as an example, with any folder structure
* Flat, with a folder structure such as:

src/  
 main.rs  
 clone.rs  
 ..

* **Sub-commands**, where the structure is nested, here’s [delta](https://github.com/dandavison/delta/tree/master/src/subcommands) as an example, with a folder structure such as

src/  
 cmd/  
 clone.rs  
 main.rs

You can go with a flat or nested structure using my starter project: [rust-starter](https://github.com/rusty-ferris-club/rust-starter) and use what you need, delete what you don’t need.

While you’re at it, it’s always good to split the core of your app and its interface. I find a good rule of thumb is to think about creating:

* A library
* A CLI that uses this library
* And something that helps generalize and solidify the API of this library: some other GUI (that will never exist) that might use this library

More often than not, especially in Rust, I find that this split was extremely useful and the use-cases happen to present themselves to me to use the library as a library by its own.

Typically it splits to three parts:

1. A library
2. A main workflow/runner, (e.g. workflow.rs) that is driving the CLI app
3. The CLI parts (prompting, exit handling, parsing flags, and commands) that maps into the workflow

**Flags, Options, Arguments, and Commands**

Even if we rule out CLI apps that are very graphical like Vim (uses TUI, etc.), we still have to address UI/UX concerns in our CLI, in the shape of flags, commands, and arguments that a program receives.

There are several different ways to specify and parse command line options, each with its own set of conventions and best practices.

According to the [POSIX](https://en.wikipedia.org/wiki/POSIX) standard, options are specified using a single dash followed by a single letter, and can be combined into a single argument (e.g. -abc). Long options, which are usually more descriptive and easier to read, are specified using two dashes followed by a word (e.g. --option). Options can also accept values, which are specified using an equals sign (e.g. -o=value or --option=value).

Positional arguments are the arguments that a command line application expects to be specified in a particular order. They are not prefixed with a dash, and are often used to specify required data that the application needs to function properly.

The POSIX standard also defines a number of special options, such as -h or --help for displaying a help message, and -v or --verbose for verbose output. These options are widely recognized and used by many command line applications, making it easier for users to discover and use different features.

Overall, the POSIX standard provides a set of conventions for specifying and parsing command line options that are widely recognized and followed by many command line applications, making it easier for users to understand and use different command line tools.

Or in other words, when designing the interface for a command-line app, we need to think about:

1. **Sub-commands:** Some command line applications allow users to specify sub-commands, which are essentially additional sub-applications that can be run within the main application. For example, the git command allows users to specify sub-commands such as commit, push, and pull. Sub-commands are often used to group related functionality within a single application and make it easier for users to discover and use different features.
2. Positional arguments: Positional arguments are the arguments that a command line application expects to be specified in a particular order. For example, the cp command expects two positional arguments: the source file and the destination file. Positional arguments are often used to specify required data that the application needs to function properly.
3. **Flags/options**: Flags are command line options that do not expect a value to be specified. They are often used to toggle a particular behavior or setting in the application. Flags are typically specified using a single dash followed by a single letter (e.g. -v for verbose output) or a double dash followed by a word (e.g. --verbose for verbose output). Flags can also accept optional values, which are specified using an equals sign (e.g. --output=file.txt).

To get a great library to rely on, that will get you from a simple app to a complex one without the need for migrating to another library, you can use [the clap crate](https://github.com/clap-rs/clap). Clap will carry you a long way; consider using an alternative only if you have a special requirement such as faster compile time, smaller binary, or similar.

Command::new("git")  
 .about("A fictional versioning CLI")  
 .subcommand\_required(true)  
 .arg\_required\_else\_help(true)  
 .allow\_external\_subcommands(true)  
 .subcommand(  
 Command::new("clone")  
 .about("Clones repos")  
 .arg(arg!(<REMOTE> "The remote to clone"))  
 .arg\_required\_else\_help(true),  
 )  
 .subcommand(  
 Command::new("diff")  
 .about("Compare two commits")  
 .arg(arg!(base: [COMMIT]))  
 .arg(arg!(head: [COMMIT]))  
 .arg(arg!(path: [PATH]).last(true))  
 .arg(  
 arg!(--color <WHEN>)  
 .value\_parser(["always", "auto", "never"])  
 .num\_args(0..=1)  
 .require\_equals(true)  
 .default\_value("auto")  
 .default\_missing\_value("always"),  
 ),  
 )  
 .subcommand(  
 Command::new("push")  
 .about("pushes things")  
 .arg(arg!(<REMOTE> "The remote to target"))  
 .arg\_required\_else\_help(true),  
 )  
 .subcommand(  
 Command::new("add")  
 .about("adds things")  
 .arg\_required\_else\_help(true)  
 .arg(arg!(<PATH> ... "Stuff to add").value\_parser(clap::value\_parser!(PathBuf))),  
 )  
 .subcommand(  
 Command::new("stash")  
 .args\_conflicts\_with\_subcommands(true)  
 .args(push\_args())  
 .subcommand(Command::new("push").args(push\_args()))  
 .subcommand(Command::new("pop").arg(arg!([STASH])))  
 .subcommand(Command::new("apply").arg(arg!([STASH]))),  
 )

A minimalistic alternative for clap is [**argh**](https://lib.rs/crates/argh). Why choose something else?

* clap may be too big in size for your binary, and you *really* care about binary size (we’re talking numbers like 300kb vs 50kb here)
* clap may take longer to compile (but it’s fast enough for most people)
* You might have a very simple CLI interface and appreciate code that fits half a screen for everything you need to do.

Here’s an example using argh

#[derive(Debug, FromArgs)]  
struct AppArgs {  
 /// task to run  
 #[argh(positional)]  
 task: Option<String>,  
 /// list tasks  
 #[argh(switch, short = 'l')]  
 list: bool,  
 /// root path (default ".")  
 #[argh(option, short = 'p')]  
 path: Option<String>,  
 /// init local config  
 #[argh(switch, short = 'i')]  
 init: bool,  
}

And then just:

let args: AppArgs = argh::from\_env();

**Configuration**

In most operating systems, **there are standard locations for storing configuration files** and other user-specific data. These locations are often referred to as “home folders” or “profile directories”, and are used to store configuration files, application data, and other user-specific data.

**Unix-like systems (Linux / macOS)**

The home folder is typically located at /home/username or /Users/username, and is used to store configuration files and other data that is specific to the user. The home folder is often referred to as the $HOME directory, and can be accessed using the ~ symbol (e.g. ~/.bashrc).

Within the home directory, there is often a .config folder (also known as the "configuration directory") that is used to store configuration files and other data that is specific to the user. The .config folder is a “dot” directory, so if you don’t see it, use a terminal. An application might store its configuration files in a subdirectory of the .config folder, such as ~/.config/myapp

On Windows, the home folder is typically located at C:\\Users\\username, and is used to store configuration files and other data that is specific to the user. The home folder is often referred to as the "user profile" directory, and can be accessed using the %USERPROFILE% environment variable (e.g. %USERPROFILE%\\AppData\\Roaming).

To hide all of this complexity, you can use [the dirs crate.](https://crates.io/crates/dirs)

dirs::home\_dir();  
// Lin: Some(/home/alice)  
// Win: Some(C:\\Users\\Alice)  
// Mac: Some(/Users/Alice)

dirs::config\_dir();  
// Lin: Some(/home/alice/.config)  
// Win: Some(C:\\Users\\Alice\\AppData\\Roaming)  
// Mac: Some(/Users/Alice/Library/Application Support)

**Config Files**

Reading configuration content is easy with serde because it makes it a 3 stage process:

1. “Shape” your configuration as a struct
2. Pick a format, and include a the required serde features (e.g. yaml)
3. Deserialize your configuration (serde::from\_str)

Most times that is more than enough and makes for a simple, maintainable code that can also be evolved by evolving the shape of your struct.

impl Config {  
 pub fn load(path: &Path)->Result<Self>{  
 Ok(serde::from\_str(&fs::read\_to\_string(path)?)?)  
 }  
..  
}

Some use cases call for “upgrading” configuration loading in two possible ways:

1. Local vs. global configuration, and their relationships. That is, read the current folder config file, and “bubble up” the folder hierarchy, completing the rest of missing configuration with each new configuration file you find until you get to the user-global configuration file residing at something like ~/.your-app/config.yaml
2. Layered configuration inputs. That is, reading from a local config.yaml but, if a certain value was provided via an environment flag or CLI flag, override what was found in config.yaml with that value. This calls for a library that can provide alignment of configuration keys for various formats: YAML, CLI flags, environment variables, and so on.

While I strongly urge you to keep things simple and evolve (just use serde loading), I found both of these libraries really robust at reading layered configuration, and do pretty much the same thing:

* <https://lib.rs/crates/figment>
* <https://github.com/mehcode/config-rs>

**Colors, Styling, and the Terminal**

In today’s world, it’s become completely OK to *express yourself* through styling in the terminal. That means colors, sometimes RGB colors, emoji, animation, and more. Using colors, unicode, risks moving away from being compatible with a “traditional” Unix terminal, but most libraries can detect and downgrade the experience as needed.

**Colors**

If you’re not using any other terminal UX library, [owo-colors](https://github.com/jam1garner/owo-colors" \t "_blank) is great, minimal, resource-friendly, and fun:

use owo\_colors::OwoColorize;  
fn main() {  
 // Foreground colors  
 println!("My number is {:#x}!", 10.green());  
 // Background colors  
 println!("My number is not {}!", 4.on\_red());  
}

If you’re using something like [dialoguer](https://github.com/console-rs/dialoguer) for prompts, it’s worth examining what it uses for colors. In this case, it uses [console](https://github.com/console-rs/console) for terminal manipulation. With console you can style it in this way:

use console::Style;  
let cyan = Style::new().cyan();  
println!("This is {} neat", cyan.apply\_to("quite"))

A bit different, but not too different.

**Emoji**

Thinking about emojis: they’re a form of [expression](https://en.wikipedia.org/wiki/List_of_emoticons). So, 🙂, :-) and [smiling] are all the same expression but different *medium*. You want the emoji where you’ve got good unicode support, the text smiley on text terminals with no unicode, and the verbose smiling when you want searchable text, or for a more accessible and readable output for visually impaired.

One other tip to remember is that Emoji can look different in different terminals. On Windows, you have the old terminal cmd.exe and Powershell, and they are radically different in how they render Emoji from Linux and macOS terminals (while Linux and macOS Emoji rendering are pretty close).

For that matter, it’s best to abstract your literal emojis under variables. It can be just a bunch of literals with your own switching logic or something fancier with a fmt::Display implementation.

You might want to switch based on a matrix of requirements:

* Operating system
* Terminal feature support (unicode, istty)
* User requested value (did they ask for no emojis specifically?)

There’s a [great implementation of that](https://github.com/console-rs/console/blob/ab9cd9dd7e65c11e050988e9a706a991ed7749cc/src/utils.rs#L691) idea (though not as extensive) in console

use console::Emoji;  
println!("[3/4] {}Downloading ...", Emoji("🚚 ", ""));  
println!("[4/4] {} Done!", Emoji("✨", ":-)"));

**Tables**

One of the most flexible libraries for printing and formatting tables is [tabled](https://github.com/zhiburt/tabled). What makes a table printing library flexible?

* Supporting “free form data” — just a set of rows and column names
* Supporting typed records through serde so you just give it a bunch of typed element in a Vec
* Formatting and shaping: alignment, spacing, spanning, and more
* Colors support — this one isn’t that easy, for when you’re calculating a table layout, you need to consider ANSI codes which makes a string byte-by-byte longer and hard to predict
* And much more

tabled does all that, and is great. If you’re looking for printing table results, or just table-layout results (such as page-formatting results), don’t look for anything else, this is it.



**Prompts**

[dialoguer](https://lib.rs/crates/dialoguer) is the most widely used prompting library out there at the moment, and is rock solid. It has almost all of the different prompts you would expect from a versatile CLI app such as checkbox, option selects, and fuzzy selects.

let items = vec!["Item 1", "item 2"];  
let selection = FuzzySelect::with\_theme(&ColorfulTheme::default())  
 .items(&items)  
 .default(0)  
 .interact\_on\_opt(&Term::stderr())?;  
  
match selection {  
 Some(index) => println!("User selected item : {}", items[index]),  
 None => println!("User did not select anything")  
}

It has one major issue — [which is a lack of a testability story](https://github.com/console-rs/dialoguer/issues/95). That is, if you want to test your code and it depends on it, you have a hard dependency on the terminal (your tests will hang).

What you would expect is to have some kind of IO abstraction facility that you can inject on tests, feed it keystrokes programmatically and verify they’ve been read and that the appropriate actions have been taken.

Another library is [inquire](https://github.com/mikaelmello/inquire) , but it too suffers from not having a testing story, and you can see how complex such a thing might be in an [issue I'm tracking.](https://github.com/mikaelmello/inquire/issues/71)

The good news is that you have a pretty good testing story with the *much less popular* library [requestty](https://lib.rs/crates/requestty" \t "_blank) though, either way when injecting that IO abstraction layer, you have to also think about mutability and ownership:

pub struct Prompt<'a> {  
 config: &'a Config,  
 events: Option<TestEvents<IntoIter<KeyEvent>>>,  
 show\_progress: bool,  
}  
  
impl<'a> Prompt<'a> {  
 pub fn build(config: &'a Config, show\_progress: bool, events: Option<&RunnerEvents>) -> Self {  
 events  
 .and\_then(|evs| evs.prompt\_events.as\_ref())  
 .map\_or\_else(  
 || Prompt::new(config, show\_progress),  
 |evs| Prompt::with\_events(config, evs.clone()),  
 )  
 }  
...

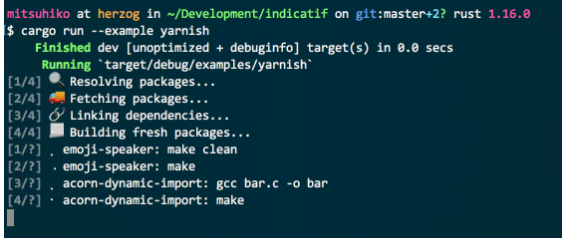
It isn’t a pretty sight but it makes for a well tested CLI interaction flow.

**What other options do you have to handle testing?**

* Trust the rock-stable state of *dialoguer* and just not test the interaction parts of your app
* Test the interaction via black-box testing (more on testing later). You can get pretty far with this approach in terms of your ROI on testing
* Build your own test rig, with a switchable UI interaction layer, where you completely replace it in test time with something that replays action (again, your own custom code). This does mean that the real code dealing with prompts and selects will never be tested, as little code as that may be

**Status and Progress**

[indicatif](https://docs.rs/indicatif/latest/indicatif/) is the gold standard Rust library for status and progress bars. There’s only one great library and that’s a good thing because we’re not stuck in the paradox of options, so use this one!



**Operability**

There are mostly two flavors of logging for operability in Rust and you can use both or one of those:

* **Logging** — and what evolved as a standard: <https://lib.rs/crates/log> where people mostly combine it with [env\_logger](https://lib.rs/crates/env_logger" \t "_blank) which is simple and really easy to use.

env\_logger::init();  
info!("starting up");

$ RUST\_LOG=INFO ./main[2018-11-03T06:09:06Z INFO default] starting up

* **Tracing** — for this, you need to use the one [tracing](https://lib.rs/crates/tracing) crate and the ecosystem that Rust has (luckily there’s just one!). You can start with [tracing-tree](https://lib.rs/crates/tracing-tree) but you can also later decide to plug in telemetry and third-party SDKs as well as print flame graphs as you would expect from a tracing infrastructure

server{host="localhost", port=8080}  
 0ms INFO starting  
 300ms INFO listening  
 conn{peer\_addr="82.9.9.9", port=42381}  
 0ms DEBUG connected  
 300ms DEBUG message received, length=2  
 conn{peer\_addr="8.8.8.8", port=18230}  
 300ms DEBUG connected  
 conn{peer\_addr="82.9.9.9", port=42381}  
 600ms WARN weak encryption requested, algo="xor"  
 901ms DEBUG response sent, length=8  
 901ms DEBUG disconnected  
 conn{peer\_addr="8.8.8.8", port=18230}  
 600ms DEBUG message received, length=5  
 901ms DEBUG response sent, length=8  
 901ms DEBUG disconnected  
 1502ms WARN internal error  
 1502ms INFO exit

Tracing lets you instrument your code very easily by decorating functions:

#[tracing::instrument(level = "trace", skip\_all, err)]  
pub fn is\_archive(file: &File, fval: &[String]) -> Result<Option<bool>>

And you can have the option for capturing arguments, return values and errors from the function automatically.

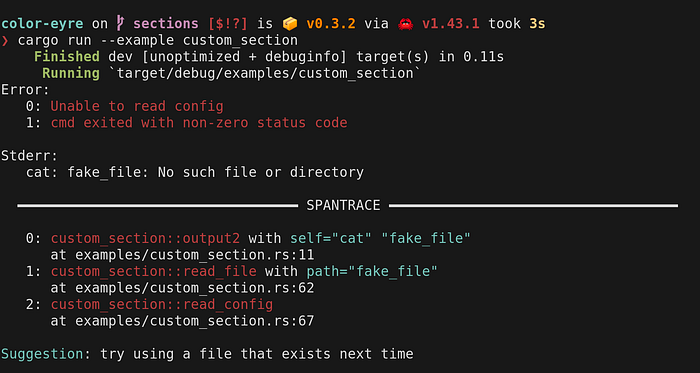
**Error Handling**

This is a big one in Rust. Simple because errors went through a big stage of evolution. There have been a few libraries that took off, then died out, and then a few more that took off and died out.

All in all, it was a fantastic process. Between each cycle the Rust ecosystem had real learnings done, and improvements made, and today we have a few libraries that are really great.

The down side of it, is that depending on the code you’ll be reading, examples here and there, and open source projects — you’ll need to make a mental note of what libraries it uses and to which era of error libraries it belongs.

So, as of writing this article, these are the libraries that I found to be perfect for a CLI app. Here too, I divide my thinking into “app errors” and “library errors” where for library errors you want to use error types that rely on the standard library and not force your users to use a specialized error library.



* App errors: [eyre](https://crates.io/crates/eyre) which is a close relative of [anyhow](https://crates.io/crates/anyhow), but has a really great error-reporting story with libraries such as [color-eyre](https://crates.io/crates/color-eyre)
* Library errors: I used to use [thiserror](https://crates.io/crates/thiserror" \t "_blank) but then moved on to [snafu](https://crates.io/crates/snafu) which I use for everything. snafu gives you all the advantages of this-error but with the ergonomics of anyhow or eyre

And then, I use libraries that improve their error reporting. Majorly I use [fs\_err](https://crates.io/crates/fs-err" \t "_blank) instead of std::fs which has the same API but more elaborate and human-friendly errors, for example:

failed to open file `does not exist.txt`  
 caused by: The system cannot find the file specified. (os error 2)

Instead of

The system cannot find the file specified. (os error 2)

**Testing**

I find **balancing** test types and strategies in Rust can be a very delicate but rewarding task. That is, Rust is safe. It doesn’t mean your code works well out of the gate, but it does mean that on top of being a statically typed language with types that guard against many programming mistakes, it’s also safe in the sense that it removes a wide array of programmer errors relating to sharing data and ownership.

I would say carefully, that my Rust code has fewer tests compared to my Ruby or JavaScript code, and is more sound.

I find that this property of Rust, also brings back blackbox testing in a big way. Because once you’ve tested some of the insides, combining modules and integrating them is rather safe by virtue of the compiler.

So all in all my testing strategy for Rust CLI apps is:

1. **Unit tests** — testing logic in functions and modules
2. **Integration tests** — per need basis, between modules and testing out complex interaction workflows
3. **Blackbox tests** — using tools like [try\_cmd](https://lib.rs/crates/trycmd" \t "_blank) to run a CLI session, provide input, capture output, and snapshot the resulting state to be approved and saved

I use snapshot testing where I can because there’s no point in left-right-coding in tests:

* insta - <https://docs.rs/insta/latest/insta/>, takes care of dev workflow, snapshotting, review, and extras such as redaction and various serialization formats for the snapshots

#[test]  
fn test\_simple() {  
 insta::assert\_yaml\_snapshot!(calculate\_value());  
}

* trycmd - <https://lib.rs/crates/trycmd>, is really robust and works well, and is fantastically simple. You write your tests as a markdown file and it will parse, run the commands that are embedded and keep track of what results should be against the result in that same markdown file, so your tests are living documentation as well

```console  
$ my-cmd  
Hello world

**Releasing and Shipping a Binary**

Over time, I’ve shaped my release workflow **into starter projects and tools**, so instead of lining out and showcasing how to build a workflow from scratch, just use those tools and projects. And if you’re curious — read their code.

If you want to have the easiest time here you can check these out:

* [rustwrap](https://github.com/rusty-ferris-club/rustwrap) — for releasing built binaries from Github releases into homebrew or npm
* [xtaskops](https://github.com/jondot/xtaskops) — for moving some logic from your CI into Rust in the form of the xtask pattern — read more in [“Running Rust Tasks with xtask and xtaskops”](https://betterprogramming.pub/running-rust-tasks-with-xtask-and-xtaskops-a2193e67dc25)
* [rust-starter](https://www.notion.so/The-Ultimate-Guide-to-Building-CLI-Apps-in-Rust-d2c4573be3bf4d6db299d52e358e5b72) — for taking the ready-made CI workflows for streamlined testing, lint, and release

If you have a proper Rust binary, you should also consider releasing to cargo bins, look at [cargo-binstall](https://github.com/cargo-bins/cargo-binstall) for more on that.