Rust CLI Project Structure

Command-line application with clap, proper error handling, and cross-platform builds.

#rust #cli #command-line #clap

PNG PDF Copy Prompt

Project Directory

```
mycli/
    Cargo.toml  Package manifest
    Cargo.lock
    LICENSE
    README.md
    .gitignore
    rust-toolchain.toml  Pin Rust version
  > src/
      main.rs  Entry point, CL…
      lib.rs  Library code fo…
      cli.rs  Clap argument d…
      error.rs  Custom error ty…
    > commands/  Subcommand impl…
        mod.rs
        init.rs  mycli init
        run.rs  mycli run
    > config/
        mod.rs
        settings.rs  Config file par…
  > tests/  Integration tes…
      cli_tests.rs  End-to-end CLI …
```

Why This Structure?

Rust CLIs are fast, reliable, and compile to single binaries. This structure uses `clap` with derive macros for declarative argument parsing. The `lib.rs` pattern allows testing business logic separately from the CLI layer.

Key Directories

`src/main.rs` - Entry point, minimal—delegates to lib

`src/cli.rs` - Clap structs with derive macros

`src/commands/` - One module per subcommand

`src/error.rs` - thiserror-based error types

Clap Derive Pattern

```
// src/cli.rs
use clap::{Parser, Subcommand};

#[derive(Parser)]
#[command(name = "mycli", version, about)]
pub struct Cli {
    #[command(subcommand)]
    pub command: Commands,

    #[arg(short, long, global = true)]
    pub verbose: bool,
}

#[derive(Subcommand)]
pub enum Commands {
    Init { name: String },
    Run { config: Option },
}
```

Getting Started

1. `cargo new mycli`
2. `cargo add clap --features derive`
3. `cargo add anyhow thiserror`
4. Define CLI in `src/cli.rs`
5. `cargo build --release`

When To Use This

- Building fast, single-binary CLI tools
- Commands with subcommands
- Need cross-platform distribution
- Want compile-time argument validation
- Performance-critical CLI applications

Trade-offs

`Compile times` - Rust compilation is slower than Go/Python

`Learning curve` - Ownership and lifetimes take time

`Binary size` - Default builds are larger (strip + LTO helps)

Naming Conventions

`Modules` - snake_case (commands/init.rs)

`Types` - PascalCase (Cli, Commands, Config)

`Functions` - snake_case (run_command, parse_config)

Testing Strategy

`Unit tests` - In-module #[cfg(test)] blocks

`Integration tests` - tests/ directory for CLI invocation

`assert_cmd` - Crate for testing CLI output