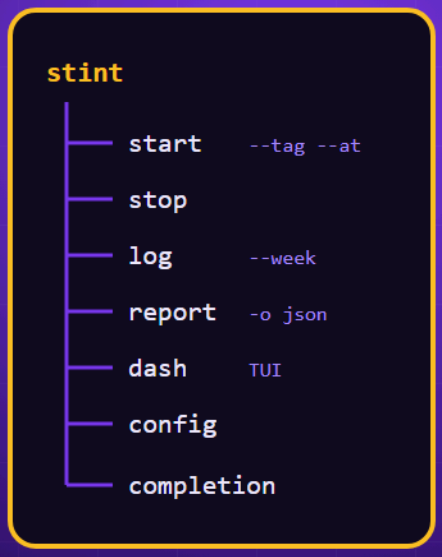
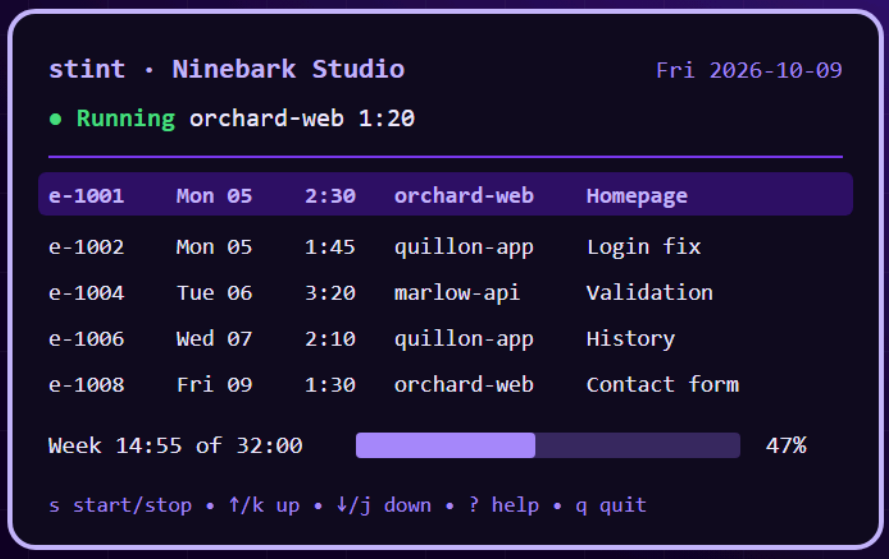


```
$ stint report --week

PROJECT            ENTRIES     TIME        AMOUNT
internal                 1     0:40         $0.00
marlow-api               2     4:45       $600.00
orchard-web              3     5:35       $546.25
quillon-app              2     3:55       $440.00
TOTAL                    8    14:55     $1,586.25

$ stint log -o json | jq length
8
```







A COMPLETE, PRACTICAL GUIDE

Building CLI & TUI Tools in Go

Professional command-line apps and terminal UIs with Cobra, Bubble Tea, and Lip Gloss

Flags · Config · JSON output · Exit codes · Pipes · Signals · Color
TUIs · Golden tests · Completion · GoReleaser · Self-update · Plugins

DESIGN · BUILD · TEST · SHIP

THENEXBOOKS

Building CLI and TUI Tools in Go

Professional Command-Line Apps and Terminal UIs
with Cobra, Bubble Tea, and Lip Gloss
From flags to tested, released, self-updating binaries

First Edition

THENEXBOOKS

Building CLI and TUI Tools in Go
Professional Command-Line Apps and Terminal UIs with Cobra, Bubble Tea, and Lip Gloss

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Go, its tools, and the libraries discussed in this book change over time; APIs, flags, configuration formats, and default behaviors can differ between versions. Always consult the official documentation for the versions you use.

Go and the Go gopher are trademarks of Google LLC. Windows, PowerShell, winget, and Windows Terminal are trademarks of Microsoft Corporation. macOS is a trademark of Apple Inc. Linux is a registered trademark of Linus Torvalds. GitHub and GitHub Actions are trademarks of GitHub, Inc. Homebrew, Scoop, GoReleaser, Cobra, Bubble Tea, Bubbles, Lip Gloss, Huh, Charm, and all other product names, libraries, and services mentioned are trademarks or names of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by Google LLC, Charmbracelet, Inc., the Cobra or GoReleaser maintainers, or any other company or project mentioned.

Ninebark Studio and its time tracker stint, used as the running example in this book, are fictional, as are the studio's team members, its clients (Orchard Lane Clinic, Quillon Bikes, Marlow Tea Rooms, Pellham Flowers), and all sample data. Module paths use the reserved example.com domain, release addresses use the reserved .example top-level domain, and the GitHub organization name ninebark-studio is used for illustration only.

Preface

Some of the most loved software in the world runs in a terminal. Developers reach for `git`, `kubectl`, `gh`, and `docker` dozens of times a day, and they notice immediately when a tool feels wrong: a flag that means something different in every subcommand, an error message that says nothing useful, a table that turns into garbage when piped into another program, colors that appear in a log file, or a progress bar that never cleans up after itself. A command-line tool is a user interface, and its users are unusually demanding.

Go is an excellent language for building these tools. It compiles to a single static binary with no runtime to install, cross-compiles for every major operating system from one machine, starts instantly, and has a mature ecosystem: Cobra for command trees, Bubble Tea, Bubbles, and Lip Gloss for rich terminal interfaces, and GoReleaser for shipping. But knowing the libraries is not the same as building a tool people enjoy using, trust in scripts, and can install with one command.

This book closes that gap. We take one realistic tool from a first prototype built on the standard `flag` package to a polished, tested, released, self-updating application with an interactive dashboard and a plugin system. Along the way we cover the conventions that make a CLI feel professional, and we explain the reasoning behind each one, so you can make good decisions in your own tools.

Who This Book Is For

- **Go developers** who can write and test ordinary Go programs and now want to build command-line tools that feel professional.
- **Platform, DevOps, and SRE engineers** who write internal tools that colleagues run every day, in terminals and in CI pipelines.
- **Backend developers** who maintain admin CLIs, migration tools, or developer utilities next to their services.
- **Open-source maintainers** who want their tool to install cleanly on Linux, macOS, and Windows and to stay stable as it grows.

We assume you know Go itself: structs, interfaces, errors, packages and modules, goroutines, and `go test`. We do not re-teach the language; we show how to use it to build tools.

What You Will Learn

- What users expect from a command-line tool, and how to design commands, arguments, flags, and help text that meet those expectations.
- How to build a command tree with Cobra, and how to structure a CLI project so that every command is testable without touching the real terminal.
- How to layer configuration from defaults, files, environment variables, and flags, with clear precedence that users can inspect.

- How to print for humans and for machines: aligned tables, stable JSON, and CSV, and how to design errors and exit codes that scripts can rely on.
- How to work with stdin, stdout, pipes, and prompts, and how to handle Ctrl+C and termination signals gracefully.
- How to use color and style responsibly, respecting `NO_COLOR` and non-terminal output, and how to show progress for long-running work.
- How to build interactive terminal UIs with Bubble Tea v2, Bubbles, and Lip Gloss v2, from the Elm architecture to a complete dashboard with a live timer.
- How to test commands in-process, with golden files, with end-to-end `testscript` scripts, and how to test TUIs with update tests, frame snapshots, and teatest.
- How to add shell completion and man pages, cross-compile, stamp versions, release with GoReleaser, publish to Homebrew, Scoop, and winget, update a running binary safely, and support plugins.

How This Book Is Organized

Part I — Foundations of a Professional CLI (Chapters 1–5) explains what makes a great command-line tool, builds a first prototype with the standard library, covers command and flag design, introduces Cobra, and sets up a project structure that keeps every command testable. **Part II — Input, Output, and Behavior** (Chapters 6–10) covers configuration and precedence, output formats, errors and exit codes, pipes and prompts, and signals and cancellation. **Part III — Terminal Craft** (Chapters 11–15) covers color and style, progress indicators, logging and diagnostics, files and cross-platform behavior, and shell completion and man pages. **Part IV — Interactive TUIs with Bubble Tea** (Chapters 16–20) teaches the Elm architecture, layout with Lip Gloss, the Bubbles components, and builds stint's dashboard, including asynchronous work and external editors. **Part V — Testing CLIs and TUIs** (Chapters 21–24) covers in-process command tests, golden files and `testscript`, testing Bubble Tea programs, and quality gates in continuous integration. **Part VI — Shipping and Extending** (Chapters 25–28) covers versioning and cross-compiling, releasing with GoReleaser and package managers, self-update, and plugins, and ends with a production checklist. The appendices provide a cheat sheet, templates and reference tables, a production checklist, and a glossary.

Each chapter ends with **Try It Yourself** exercises and a **Key Takeaways** box for quick review.

The Running Example: stint

Throughout the book we build **stint**, the time tracker of **Ninebark Studio**, a small fictional product studio that builds websites and mobile apps for clients such as Orchard Lane Clinic, Quillon Bikes, and Marlow Tea Rooms. Developers start and stop timers from the terminal, list their entries, and turn the week into a billable report; designers prefer an interactive dashboard with a live timer and a weekly progress bar; the studio lead wants CSV for the accountant; and a contractor on Windows finds every

portability bug. stint is small enough to understand completely, yet it needs everything a real tool needs: subcommands, configuration, several output formats, careful errors, pipes, signals, color, a TUI, tests, completion, releases, updates, and extensions. The sample data is small and fixed, so every number in a report can be checked by hand.

Versions and Conventions

The code targets Go 1.26 or later and was tested with Go 1.27.1. It uses Cobra 1.10, Bubble Tea v2, Bubbles v2, and Lip Gloss v2 (imported from `charm.land`), BurntSushi's TOML library, `golang.org/x/term`, and `testscript` from `go-internal`; exact versions are listed in Chapter 1 and Appendix B. The Charm libraries changed significantly in their v2 releases, and many tutorials online still show v1 code; we point out the differences where they matter. Commands, outputs, and test results in this book come from real runs. Where something cannot be run on a single development machine, such as a GoReleaser release or a package-manager installation, we say so and show output "similar to" what you will see.

> **TIP**
> A recommended practice or a technique that makes your tool easier to use, test, or maintain.

> **NOTE**
> Background information, version differences, or a useful aside.

> **WARNING**
> A common mistake, a portability trap, or something that can break your users' scripts.

Shell sessions are shown with a `$` prompt for bash or zsh and `PS>` for PowerShell. Go code is formatted with `gofmt`; long lines are wrapped to fit the page. File captions show paths relative to the module root, such as `internal/cli/root.go`. Terminal UI screens are printed as plain-text frames rendered by tests at a fixed size, with colors described in the text.

Contents

Preface

PART I — FOUNDATIONS OF A PROFESSIONAL CLI

1 What Makes a Great Command-Line Tool: Conventions, Expectations, and Meet stint

2 From os.Args to the flag Package: A First Prototype

3 Designing Commands, Arguments, and Flags: Names, Defaults, and Help Text

4 Cobra Fundamentals: Commands, Flags, Argument Validation, and the Command Tree

5 Structuring a CLI Project: Packages, the App Struct, and a Testable main

PART II — INPUT, OUTPUT, AND BEHAVIOR

6 Configuration: Files, Environment Variables, Flags, and Precedence

7 Output for Humans and Machines: Tables, JSON, and CSV

8 Errors, Exit Codes, and Messages Users Can Act On

9 stdin, stdout, and Pipes: Filters, Imports, and Safe Prompts

10 Signals, Context, and Cancellation: Ctrl+C, Timeouts, and Graceful Shutdown

PART III — TERMINAL CRAFT

11 Color and Style with Lip Gloss: NO_COLOR, TTY Detection, and Accessibility

12 Progress Bars, Spinners, and Status Lines for Long-Running Work

13 Logging and Diagnostics: log/slog, --verbose, and Bug Reports

14 Files, Paths, and Cross-Platform Behavior: Config Directories, Atomic Writes, Locks, and Windows

15 Shell Completion, Help Topics, and Man Pages

PART IV — INTERACTIVE TUIS WITH BUBBLE TEA

16 The Elm Architecture in Bubble Tea v2: Model, Update, View, and Commands

17 Layout and Styling for TUIs with Lip Gloss: Boxes, Joins, Widths, and Resizing

18 Bubbles Components: Tables, Lists, Text Inputs, Spinners, Progress Bars, and Help

19 Building the stint Dashboard: Screens, Focus, Key Maps, and a Live Timer

20 Async Work in TUIs: Commands, Ticks, External Editors, and Forms

PART V — TESTING CLIS AND TUIS

21 Testing Commands In-Process: Fake Clocks, Fake Environments, and Captured Output

22 Golden Files and End-to-End Tests with testscript

23 Testing Bubble Tea Programs: Update Tests, Frame Snapshots, and teatest

24 Quality Gates: Cross-Platform CI, the Race Detector, Fuzzing, and Linters

PART VI — SHIPPING AND EXTENDING

25 Versioning and Cross-Compiling: ldflags, Build Info, GOOS/GOARCH, and Reproducible Builds

26 Releasing with GoReleaser: Archives, Checksums, Homebrew, Scoop, and winget

27 Self-Update and Update Notices: Release Manifests, Checksums, and Replacing a Running Binary

28 Plugins and Extensibility, and a Production Checklist for CLI Tools

APPENDICES

A CLI and TUI Cheat Sheet

B Templates and Reference

C Production Checklist for CLI and TUI Tools

D Glossary

PART I

Foundations of a Professional CLI

What users expect from a command-line tool, a first prototype with the standard library, designing commands, arguments, and flags, building a command tree with Cobra, and a project structure that keeps every command testable.

1 · What Makes a Great Command-Line Tool
2 · From os.Args to the flag Package
3 · Designing Commands, Arguments, and Flags
4 · Cobra Fundamentals
5 · Structuring a CLI Project

CHAPTER 1 · PART I — FOUNDATIONS OF A PROFESSIONAL CLI

What Makes a Great Command-Line Tool: Conventions, Expectations, and Meet stint

Before we write a single flag, we look at what users silently expect from every command-line tool, why those conventions exist, and how they shape the design of stint, the time tracker we build throughout this book. We finish with a tour of the finished tool and a working Go setup.

Why Command-Line Tools Still Matter

Developers and operators spend much of their day in terminals for good reasons. A command can be repeated exactly, pasted into a chat message, saved in a script, run on a remote server over SSH, and executed by a CI pipeline with no screen at all. A command is also a contract: `git log --oneline -5` means the same thing today, next month, and on a colleague's laptop. That is why the tools developers rely on most, such as `git`, `kubectl`, `gh`, `docker`, and `go` itself, are command-line tools first.

Internal tools follow the same path. A platform team writes a small program to rotate certificates, a backend team writes an admin tool to replay failed jobs, a studio writes a time tracker. They begin as a weekend script; then someone pipes the output into another program, runs it from cron, or tries it on Windows. At that moment the tool stops being a script and becomes a product, with users who have expectations they rarely state out loud.

Go fits that moment well: one binary with no runtime to install, quick startup, easy cross-compiling, and a mature ecosystem (Cobra, Bubble Tea, Lip Gloss, GoReleaser). The first step is to understand what "professional" means to the person typing the command.

What Users Expect from a Command-Line Tool

Most CLI conventions are not written down in one place. They come from decades of Unix tools, the POSIX utility guidelines, GNU's long options, and the habits of popular modern tools. Users learn them by osmosis and notice immediately when a tool breaks one. Here are the expectations we design stint around, with the reason for each; the reason tells you when a convention can bend.

Help that is always one flag away

Every command and subcommand should answer `-h` and `--help` with a short description, a usage line, examples, and a list of flags. Users reach for help in the middle of a task, so it must be fast to scan: what does this command do, what must I type, and what does a typical call look like? Examples are the most-read part of any help text, because people copy them.

A version you can report

`--version` (and often a `version` subcommand) tells users which build they are running. This sounds trivial until you receive a bug report. "It crashes when I run report" is hard to act on; "stint 1.0.0 (commit 3f9c2ab, built 2026-10-08) windows/amd64 crashes when I run report" points at a specific build on a specific platform. We stamp the version into the binary at build time in Chapter 25.

Predictable flags and arguments

Users expect GNU-style flags: long options such as `--output json` or `--output=json`, short options such as `-o json` for the most frequent choices, boolean short flags that can be bundled (`-vy`), flags that may appear before or after positional arguments, and `--` to mean "everything after this is an argument, even if it starts with a dash". They also expect the same flag to mean the same thing in every subcommand: if `-p` means "project" in one command, it must not mean "port" in another. Consistency lets people guess correctly, and guessing correctly is what makes a tool feel easy.

Data on stdout, everything else on stderr

A process has two output streams, and they have different jobs. **Standard output** is for the result the user asked for: the table, the JSON, the CSV. **Standard error** is for everything about the run: error messages, warnings, prompts, progress bars, and debug logs. The split exists so that redirection and pipes work. When Kwame runs `stint report -o json > week.json`, the file must contain only JSON, even if stint also printed a warning. When a command fails, nothing should appear on stdout, so a script that captures stdout does not accidentally process half a result. We can see the split with the finished stint:

Git Bash

```
$ stint report --from 2026-10-05 --to 2026-10-11 > report.txt 2> err.txt
$ echo $?
0
$ wc -c report.txt err.txt
282 report.txt
  0 err.txt
282 total
$ stint start nope > out.txt 2> err.txt
$ echo $?
3
$ wc -c out.txt err.txt
 0 out.txt
33 err.txt
33 total
```

The successful report wrote 282 bytes of table to stdout and nothing to stderr. The failed `start` wrote nothing to stdout and a 33-byte error message to stderr. A script can rely on both facts.

Exit codes that mean something

Every process ends with an exit status, and shells and CI systems read it. Zero means success, anything else means failure. That single bit is what makes `stint stop && git push` safe, and what makes a CI step turn red. Many tools go further and give specific failures specific codes. A common convention is 2 for usage mistakes (a misspelled flag, a missing argument), and POSIX shells report 128 plus the signal number when a process is killed by a signal, so 130 means "interrupted by Ctrl+C". stint uses a small, documented table: 0 success, 1 unexpected failure, 2 usage error, 3 not found, 4 conflict, 130 interrupted. Chapter 8 explains how we map errors to those codes, and why we never renumber one once it is published.

Quiet success

The old Unix "rule of silence" says that a program with nothing surprising to say should say nothing. We take a slightly softer position: a command that changes something should confirm it in one short line (`Stopped e-1009 orchard-web at 15:00 (1:20).`), and a command that only reads should print the data and nothing else. No banners and no "Loading configuration...": chatty output trains users to ignore output, and then they miss the one line that mattered.

Composability

The most valuable property of a CLI is that it combines with other tools. That requires output a machine can read reliably: a stable JSON shape, CSV with a header row, one ID per line for `xargs`. It also requires accepting input from pipes, using `-` to mean standard input, and behaving correctly when the reader goes away early, as in `stint log | head -2`. Human-friendly tables are the default when a person is looking; machine-friendly formats are one flag away when a program is reading.

Respect for the terminal

Color, bold text, progress bars, and full-screen interfaces are great in an interactive terminal and terrible in a log file. A well-behaved tool checks whether its output is a terminal (a TTY) before it adds escape codes, honors the `NO_COLOR` environment variable, which by the convention at no-color.org disables color when it is set to any non-empty value, and offers `--color` or `--no-color` for explicit control. Interactive prompts follow the same rule: a tool must not wait for an answer when nobody is there to type one. Chapter 11 covers color in depth, and Chapter 9 covers prompts.

Safety for destructive actions

Commands that delete or overwrite data should ask first when a person is at the keyboard, offer `--yes` (or `--force`) for scripts, and offer `--dry-run` when the effect is large. `stint delete e-1003` asks for confirmation; `stint delete e-1003 --yes` does not; and `stint delete` run from a pipe refuses to guess.

> **TIP**
>
> When you are unsure how a flag or behavior should work, check what `git`, `kubectl`, and `gh` do. You do not have to copy them, but matching the tools your users already know removes a whole class of surprises.

CLI, TUI, or GUI?

A **command-line interface** (CLI) takes its input from arguments, flags, environment variables, and standard input, prints a result, and exits. A **terminal user interface** (TUI) takes over the terminal screen, redraws it many times per second, and reacts to key presses, like `htop`, `lazygit`, or `vim`. A **graphical user interface** (GUI) runs in its own window or in a browser. Each has a natural home:

Interface	Strengths	Weaknesses	Best for
CLI	Scriptable, composable, repeatable, works over SSH and in CI	Must remember commands; poor for browsing	Actions and queries with clear inputs
TUI	Interactive and visual, keyboard-driven, still runs in a terminal	Not scriptable; harder to test	Monitoring, browsing, live status
GUI	Discoverable, rich visuals, mouse-friendly	Heavy to build and ship; not scriptable	Occasional users, complex editing

The interesting case is a tool that offers both a CLI and a TUI. Our rule is simple: **every capability must be available as a plain command first**, and the TUI is a convenience layered on top. That way scripts and CI never need the TUI, and the TUI can reuse the same domain code. stint follows this rule: `stint start`, `stop`, `log`, and `report` do all the work, and `stint dash` is a full-screen dashboard for the people who like to watch a timer tick.

Meet Ninebark Studio and stint

Our running example comes from **Ninebark Studio**, a fictional 14-person digital product studio that builds websites and mobile apps for small clients. The team is remote and spread across several time zones. The studio bills most work by the hour, so tracking time accurately is not optional: it is how the studio gets paid. Its platform engineer wrote **stint**, a small time tracker, and in this book we (the people who maintain stint) turn it into a polished, tested, released tool.

Four people shape stint's requirements, and they will appear as design reviewers throughout the book:

- **Odile**, the studio lead, wants accurate billable reports per client and a CSV file she can send to the accountant at the end of the month.
- **Kwame**, a backend developer, lives in the terminal. He wants short commands, JSON output he can pipe into other tools, exit codes he can trust in scripts, and shell completion.

- **Hana**, a designer, prefers something she can look at: a dashboard with a live timer, color, and a weekly progress bar.
- **Ruben**, a contractor, works on Windows. He finds every path bug, every console encoding problem, and every assumption that only holds on Linux or macOS.

Their needs map directly onto the topics of this book:

Who	Wants	Where we deliver it
Odile	Billable report, CSV export, correct rounding	Chapters 7 and 8
Kwame	JSON, pipes, exit codes, completion	Chapters 7, 8, 9, and 15
Hana	Color, dashboard, live timer	Chapters 11 and 16–20
Ruben	Windows paths, consoles, installers	Chapters 14, 25, and 26

The studio's clients are fictional too: Orchard Lane Clinic (a clinic website, project `orchard-web`), Quillon Bikes (a rider mobile app, `quillon-app`), and Marlow Tea Rooms (a booking API, `marlow-api`). Internal work such as planning meetings goes to the non-billable `internal` project. Each project has an hourly rate in US dollars, and stint rounds billable entries up to the nearest 15 minutes before pricing them, which is a common studio policy and a good source of off-by-one bugs.

A Tour of the Finished stint

The best way to understand where we are going is to use the finished tool. Everything below is real output from the stint we build in this book, run with its built-in sample data: four projects and eight entries in the week from Monday 2026-10-05 to Sunday 2026-10-11. We use explicit dates so that the output does not depend on the day you read this.

Help

The top-level help groups commands by task, shows real examples, and lists the global flags that every command accepts:

Terminal

```
$ stint --help
stint tracks the time you spend on client projects.

Start a timer, stop it, and turn the week into a billable report.
Data lives in one JSON file; settings come from flags, STINT_*
environment variables, a TOML config file, and defaults, in that order.

Usage:
  stint [command]

Examples:
  stint start orchard-web "Contact form" --tag design
  stint stop
```

```
  stint report --week
  stint log --from 2026-10-05 -o json | jq '.[].minutes'

Tracking time:
  add         Record time you forgot to track
  dash        Open the interactive dashboard
  start       Start a timer on a project
  status      Show the running timer, if any
  stop        Stop the running timer

Reports and data:
  delete      Delete a time entry
  log         List time entries
  projects    List or add projects
  report      Summarize time and billable amounts per project

Setup:
  completion  Generate the autocompletion script for the specified shell
  config      Show and change settings
  help        Help about any command
  init        Create the data file (optionally with sample data)
  version     Print version information

Flags:
      --color string       color output: auto, always, or never
      --config string      config file (default: <user config dir>/stint/config.toml)
      --data-file string   data file (overrides data_file)
  -h, --help               help for stint
      --no-color           disable color (same as --color=never)
  -o, --output string      output format: table, json, or csv
  -v, --verbose            log debug details to stderr
      --version            version for stint

Use "stint [command] --help" for more information about a command.
```

The long description explains the mental model in three lines, the examples show the most common commands, and the groups separate daily commands from occasional ones. Chapter 3 explains this text. This is the core command set; later chapters add `import`, `track`, `self-update`, `plugins`, a `formats` help topic, and a `--log-format` flag, so the final help is longer.

Creating data and listing projects

A new user starts with `stint init`. The `--sample` flag fills the data file with Ninebark's sample week, which is what we use throughout the book:

Terminal

```
$ stint init --sample --data-file stint.json
Created stint.json with 4 projects and 8 entries.
$ export STINT_DATA_FILE=stint.json
$ stint projects
CODE         CLIENT              NAME                 RATE  BILLABLE
internal     Ninebark Studio     Studio operations    $0.00  no
```

```
marlow-api   Marlow Tea Rooms     Booking API        $120.00  yes
orchard-web  Orchard Lane Clinic  Clinic website       $95.00  yes
quillon-app  Quillon Bikes        Rider mobile app   $110.00  yes
```

Two conventions are visible already. Settings can come from a flag (`--data-file`) or an environment variable (`STINT_DATA_FILE`), and the table aligns money to the right so that the decimal points line up. Running `stint init --sample` a second time fails with `stint: stint.json already exists (use --force to overwrite)` and exit code 4: the error says what happened and what to do about it.

The log

Terminal

```
$ stint log --from 2026-10-05 --to 2026-10-11
ID      DATE        START  END    TIME  PROJECT      NOTE
e-1001  2026-10-05  09:00  11:30  2:30  orchard-web  Homepage hero layout
e-1002  2026-10-05  12:15  14:00  1:45  quillon-app  Fix login crash on Android
e-1003  2026-10-05  14:00  14:40  0:40  internal     Weekly planning
e-1004  2026-10-06  09:30  12:50  3:20  marlow-api   Booking endpoint validation
e-1005  2026-10-06  13:30  15:05  1:35  orchard-web  Accessibility fixes
e-1006  2026-10-07  10:00  12:10  2:10  quillon-app  Ride history screen
e-1007  2026-10-08  09:00  10:25  1:25  marlow-api   Rate limiting
e-1008  2026-10-09  09:00  10:30  1:30  orchard-web  Contact form
8 entries, 14:55 total
```

In a real terminal the header is bold violet and the summary line is dimmed. When the output goes to a file or a pipe, stint drops the styling automatically. The `--week` flag selects the current Monday-to-Sunday week, which is what people type most days; we use explicit dates in the book only to keep the output stable.

The billable report

Terminal

```
$ stint report --from 2026-10-05 --to 2026-10-11
PROJECT      ENTRIES   TIME  BILLED     AMOUNT
internal           1   0:40    0:00      $0.00
marlow-api         2   4:45    5:00    $600.00
orchard-web        3   5:35    5:45    $546.25
quillon-app        2   3:55    4:00    $440.00
TOTAL              8  14:55   14:45  $1,586.25
```

This is Odile's favorite command. TIME is the real tracked time; BILLED is the time after rounding each billable entry up to 15 minutes; AMOUNT prices the billed time at the project's rate. Every number checks by hand. Orchard Lane Clinic had entries of 150, 95, and 90 minutes; the 95 rounds up to 105, giving 345 billed minutes, or 5:45, and 345 minutes at $95.00 per hour is $546.25. The internal project is not billable, so it bills nothing. When a test fails, a calculator gives the right answer.

Output for machines

The same data is available as JSON and CSV through the global `-o` flag. Here is the JSON report (shortened) and part of the log as CSV:

Terminal

```
$ stint report --from 2026-10-05 --to 2026-10-11 -o json
{
  "from": "2026-10-05",
  "to": "2026-10-11",
  "round_minutes": 15,
  "lines": [
    ...
  ],
  "total": {
    "project": "TOTAL",
    "entries": 8,
    "minutes": 895,
    "billed_minutes": 885,
    "amount_cents": 158625
  }
}
$ stint log --from 2026-10-06 --to 2026-10-11 --project orchard-web -o csv
id,project,start,end,minutes,note,tags
e-1005,orchard-web,2026-10-06T13:30:00Z,2026-10-06T15:05:00Z,95,Accessibility fixes,a11y
e-1008,orchard-web,2026-10-09T09:00:00Z,2026-10-09T10:30:00Z,90,Contact form,design
```

Machine output uses integers (minutes and cents) instead of formatted strings like `5:45` or `$546.25`, because programs should not have to parse human formatting. Times are RFC 3339, which every language can parse. Chapter 7 explains these choices and the promise they create: once Kwame's scripts read `amount_cents`, we cannot rename it.

Errors

Errors go to stderr, start with the program name, and come with a meaningful exit code:

Git Bash

```
$ stint start nope
stint: project "nope": not found
$ echo $?
3
$ stint lgo
stint: unknown command "lgo" for "stint"

Did you mean this?
    log

Run 'stint --help' for usage.
$ echo $?
2
```

The second error shows two small kindnesses: a suggestion for the likely typo and a hint about where to find help. Usage errors exit with code 2, so a script can tell "you called me wrong" apart from "the project does not exist" (3) or "a timer is already running" (4).

The dashboard

Finally, `stint dash` opens a full-screen terminal UI built with Bubble Tea v2. A printed page cannot show a live terminal, so throughout this book a TUI "screen" is the plain-text frame that our tests render at a fixed size with styling removed. This is the dashboard at 80 by 20 cells, with the test clock set to Friday 2026-10-09 at 15:00 UTC:

Screen: stint dash at 80x20 (TestDashboardView, styling stripped)

```
stint · Ninebark Studio                                    Fri 2026-10-09
○ No timer running

 ID       DAY     TIME         DUR    PROJECT       NOTE
──────────────────────────────────────────────────────────────────────────
 e-1001   Mon 05  09:00-11:30  2:30   orchard-web   Homepage hero layout
 e-1002   Mon 05  12:15-14:00  1:45   quillon-app   Fix login crash on Android
 e-1003   Mon 05  14:00-14:40  0:40   internal      Weekly planning
 e-1004   Tue 06  09:30-12:50  3:20   marlow-api    Booking endpoint validation
 e-1005   Tue 06  13:30-15:05  1:35   orchard-web   Accessibility fixes
 e-1006   Wed 07  10:00-12:10  2:10   quillon-app   Ride history screen
 e-1007   Thu 08  09:00-10:25  1:25   marlow-api    Rate limiting
 e-1008   Fri 09  09:00-10:30  1:30   orchard-web   Contact form



Week 2026-10-05 to 2026-10-11  14:55 of 32:00  ▌▌▌▌▌▌▌▌▌▌▒▒▒▒▒▒▒▒▒▒  47%

s start/stop • ↑/k up • ↓/j down • ? more keys • q quit
```

In the terminal the word `stint` in the title is bold violet, the "No timer running" line and the key help at the bottom are dimmed, and the column headers and the selected row are violet. The bar shows 14:55 of the 32-hour weekly target, which rounds to 47%. Pressing `s` starts a timer on the selected row's project, the status line changes to a bold green `● Running`, and a live clock counts up once per second. Part IV builds this screen from scratch.

How stint Is Built

Behind the commands is a small, layered architecture that we assemble over the first chapters. The guiding idea is that the parts that change for different reasons live in different packages:

```
cmd/stint/main.go          three lines: signals, cli.Run, os.Exit
        |
internal/cli               Cobra commands, flags, exit codes, App struct
        |            \
internal/tui               Bubble Tea dashboard (stint dash)
        |              \
```

```
internal/config            internal/output
defaults < file < env      tables, JSON, CSV, color
        \                  /
         internal/timesheet        domain: projects, entries, reports
         internal/store            load and save the JSON data file
```

The `timesheet` package knows nothing about flags, files, or terminals; it just knows how to start a timer, filter entries, and price a report. The `store` package reads and writes the data file safely. The `cli` package translates command lines into calls on the domain and results into text, and it receives everything from the outside world (streams, environment, clock) through an `App` struct so that tests can run any command in-process with fakes. Chapter 5 explains this structure in detail; for now, notice that `main` is tiny. Almost nothing in a well-structured CLI needs the real process to be tested.

Setting Up Your Environment

You need **Go 1.26 or later** (we used 1.27.1). The Charm libraries we use for the TUI part of the book, Bubble Tea v2 in particular, require Go 1.26, which is why the module's `go` line says `1.26.0`. Check your version first:

Terminal

```
$ go version
go version go1.27.1 windows/amd64
```

You also want a real terminal: Windows Terminal or a recent PowerShell on Windows, and any modern terminal emulator on Linux or macOS. On Windows, Git Bash is handy for trying the Unix pipelines in this book (`head`, `grep`, `wc`, `cat -v`); every example that matters also has a PowerShell form where the two differ.

Creating the module

We build stint step by step, so start with an empty module. The module path `example.com/stint` uses the `example.com` domain, which is reserved for documentation, so it can never clash with a real module:

Terminal

```
$ mkdir stint && cd stint
$ go mod init example.com/stint
go: creating new go.mod: module example.com/stint
go: to add module requirements and sums:
    go mod tidy
$ go mod edit -go=1.26.0
$ go get github.com/spf13/cobra@v1.10.2
go: added github.com/inconshreveable/mousetrap v1.1.0
go: added github.com/spf13/cobra v1.10.2
go: added github.com/spf13/pflag v1.0.9
```

Now add a first `main` package. It does almost nothing yet, but it proves that the toolchain, the module, and Cobra work together:

cmd/stint/main.go (first version)

```
// Command stint is Ninebark Studio's time tracker.
package main

import (
    "fmt"
    "os"

    "github.com/spf13/cobra"
)

func main() {
    root := &cobra.Command{
        Use:     "stint",
        Short:   "Track time on client projects from your terminal",
        Version: "0.1.0",
        Run: func(cmd *cobra.Command, args []string) {
            fmt.Fprintln(cmd.OutOrStdout(), "Hello from stint.")
        },
    }
    if err := root.Execute(); err != nil {
        os.Exit(1)
    }
}
```

Terminal

```
$ go mod tidy
$ go run ./cmd/stint
Hello from stint.
$ go run ./cmd/stint --help
Track time on client projects from your terminal

Usage:
  stint [flags]

Flags:
  -h, --help      help for stint
  -v, --version   version for stint
$ go run ./cmd/stint --version
stint version 0.1.0
```

Cobra already provides help, a version flag, and unknown-flag errors. Notice the `-v` shorthand that Cobra added for `--version`; the finished stint uses `-v` for `--verbose` instead, and Chapter 4 shows how that works. We keep the layout `cmd/stint/main.go` from the start, because the module will later hold more than one program (a prototype in Chapter 2, a documentation generator in Chapter 15, a plugin in Chapter 28).