Extracted from:

Powerful Command-Line Applications in Go

Build Fast and Maintainable Tools

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Powerful Command-Line Applications in Go

Build Fast and Maintainable Tools

Ricardo Gerardi

edited by Brian P. Hogan

Powerful Command-Line Applications in Go

Build Fast and Maintainable Tools

Ricardo Gerardi

The Pragmatic Bookshelf
Raleigh, North Carolina

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Brian P. Hogan
Copy Editor: Corina Lebegioara
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-696-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—December 2021

To my beloved wife Kassia. My best friend, my greatest supporter. Sadly she passed away due to cancer before seeing this book completed.

To my four incredible daughters Gisele, Livia, Elena, and Alice. The reason for everything.

Preface

Whether you're a system administrator, a network engineer, a DevOps specialist, or any other modern IT professional, you use command-line applications to automate your environment and increase your productivity. These tools play an increasingly critical role in your infrastructure and therefore require the same level of governance as other software components. In this book, you'll use the Go programming language to develop command-line applications that are maintainable, cross-platform, fast, and reliable.

Go is a modern programming language that combines the reliability provided by the compilation process with the flexibility of dynamic typing. Go's ease of use and flexibility in prototyping new ideas make it a great choice for writing command-line tools. At the same time, Go allows the implementation of more complex scenarios by providing features like type safety, cross-compilation, testing, and benchmarks.

Many popular command-line tools you use are developed with Go. These include Docker, Podman, Kubectl, Openshift CLI, Hugo, and Terraform. If you've ever wondered how you can make your own tools like these, this book will show you how.

You'll apply your knowledge of Go's basic syntax and also employ more advanced concepts to develop several command-line applications. You can use these applications to automate tasks, analyze data, parse logs, talk to network services, or address other system requirements. You'll also employ different testing and benchmarking techniques to ensure your programs are fast and reliable.

What's in This Book

In Chapter 1, Your First Command-Line Program in Go, on page ?, you'll take a quick tour through the process of developing a command-line application with Go by building a word counter. You will start with the basic implementation, add some features, and explore testing. You'll also add command-line flags and build this application for different platforms.

In Chapter 2, Interacting with Your Users, on page ?, you'll design and write a command-line tool to manage lists of to-do items in accordance with common input/output standards by applying different techniques. You'll take input from the standard input (STDIN) stream, parse command-line parameters, and define flags for your tool using the flags package. You'll use environment variables to increase the flexibility of your tools. In addition, you'll display information and results back to the user through the standard output (STDOUT) stream, and present errors using the standard error (STDERR) stream for proper error handling. Finally, you'll explore Go interfaces by applying the io.Reader interface in particular.

Next, in Chapter 3, Working with Files in Go, on page ?, you'll develop a tool to preview Markdown files using a web browser. You'll create and open files for reading and writing. You'll apply techniques to handle paths consistently across different operating systems. You'll use temporary files and apply the defer keyword to clean them up. You'll also make your tool flexible by using file templates. Finally, you'll use Go interfaces to make your code flexible, while writing and executing tests to ensure your code matches the requirements.

In Chapter 4, Navigating the File System, on page ?, you'll navigate through the file system and work with directories and file properties. You'll develop a CLI application to find, delete, and back up files according to different criteria. You'll perform common file system operations such as copying, compressing, and deleting files. You'll also log information to the screen or log files. Finally, you'll apply the concepts of table-driven testing and test helpers to write flexible and meaningful test cases for your application.

In Chapter 5, Improving the Performance of Your CLI Tools, on page ?, you'll develop a command-line tool that processes data from CSV files. Then you'll use Go's benchmarking, profiling, and tracing tools to analyze its performance, find bottlenecks, and redesign your CLI to improve its performance. You'll write and execute tests to ensure your application works reliably across the refactoring. You'll also apply Go's concurrency primitives such as goroutines and channels to ensure your application runs tasks concurrently in a safe way.

Chapter 6, Controlling Processes, on page ?, will allow you to expand your command-line applications' capabilities by executing external tools. You'll execute, control, and capture their output to develop a Continuous Integration tool for your Go programs. You'll explore different ways to execute external programs with various options such as timeouts that ensure your program doesn't run forever. You'll also handle operating system signals correctly to allow your tool to gracefully shut down.

Next, in Chapter 7, Using the Cobra CLI Framework, on page ?, you'll develop a network tool that executes a TCP port scan on remote machines by applying the Cobra CLI framework. Cobra is a popular framework that allows you to create flexible command-line tools that use subcommands compatible with the POSIX standard. You'll use Cobra to generate the boilerplate code for your application, allowing you to focus on its business logic.

In Chapter 8, Talking to REST APIs, on page ?, you'll improve your to-do application by making it available through a representational state transfer (REST) API. Then you'll develop a command-line client that interacts with this API using several HTTP methods. You'll parse JSON data and fine-tune specific parameters of your requests such as headers and timeouts. You'll also apply proper testing techniques that ensure your application works reliably without overloading web servers unnecessarily.

In Chapter 9, Developing Interactive Terminal Tools, on page ?, you'll build an interactive command-line application that uses terminal widgets to interact with the user. You'll use external packages to design and develop the interface. You'll also apply different Go concurrency techniques to manage this application asynchronously.

In Chapter 10, Persisting Data in a SQL Database, on page ?, you'll expand your interactive application by allowing users to save its data into a SQL database. You'll use Go's standard library and external packages to connect to standard databases by executing SQL operations. You'll query, insert, and delete data from databases and use a local *Sqlite3* database to persist data for your tool. You'll make this data available to the user by summarizing its content using the application interface.

And finally, in Chapter 11, Distributing Your Tool, on page ?, you'll explore several techniques to build your tool, including different build and cross-compilation options, allowing your tool to run in multiple operating systems. You'll apply build tags to change the behavior of your builds according to external conditions. You'll take a quick look at using CGO to embed C code in your Go applications. Then you'll apply techniques to package and distribute your application either as a Linux container or as source code via `go get`.

This book doesn't cover the basic syntax of the Go programming language. You should be familiar with declaring variables, types, custom types, flow control, and the general structure of a Go program. If you're starting with Go, take a look at these books and articles that do a great job explaining the language's syntax:

- *Learning Go [Jon21]*
- *Go in Action [KKS15]*
- A Tour of Go[1]
- Effective Go[2]

This book uses the Go standard library as much as possible. Go has a rich and diverse standard library that includes packages that address most of the requirements for creating command-line tools in general. By using the standard library, we benefit from Go's compatibility across different versions making the code accessible to a larger number of readers. cases, we'll use external packages when no equivalent functionality is available but we generally prefer to use the standard library even if an external package makes it easier to address a requirement. The notable exception to this rule is the Cobra CLI framework that you'll use in Chapter 7, Using the Cobra CLI Framework, on page ?, as this is a popular framework used by many developers and companies to extend Go's capabilities to manage command-line applications.

In each chapter, you'll typically develop a fully functional command-line tool. You'll start with the basic functionality, write some tests, and then add more features. At the end of each chapter, you'll find additional exercises to improve what you've learned in the chapter and practice your skills further. And you're encouraged to add more features on your own.

This book spends a fair amount of time on testing your code. In some cases, you'll see that the test examples are more intricate than the code examples. This is done for two significant reasons: as command-line tools become more critical to your infrastructure, it's essential that you ensure they work correctly; and Go provides out-of-the-box features to test and benchmark your code. You'll start by creating basic test functions. Then you'll develop more advanced concepts such as table-driven testing and dependency injection, culminating with mocking your own commands and artifacts for testing.

Finally, feel free to read this book in any order. If you have a particular interest or if one of the examples seems more appealing, feel free to jump around. Keep in mind that some chapters build on skills presented in previous chapters. A cross-reference usually points to where that concept was first discussed in the book so you can explore the topic in more detail.

1. tour.golang.org
2. golang.org/doc/effective_go.html

How to Use This Book

To best follow this book and test the code examples, you need Go 1.13 or higher installed on your machine. At the time of writing this book, the current version of Go is 1.16. You can find more information on installing Go in the official documentation.[3]

The code examples in the book use *Go modules* to manage package dependencies. For details, consult Go Modules, on page xiii. By using Go modules, you're no longer required to code under the $GOPATH directory. Modules have been available since Go 1.11 and enabled by default since 1.13.

Make sure the Go binary go is in your $PATH variable so you can execute Go commands from anywhere without prefixing the absolute path. For example, to run tests, you type go test, and to build programs, you run go build. Some examples also execute the binary version of your tools. These are typically installed in the directory $HOME/go/bin. We expect that this directory exists and is included in your $PATH variable.

You'll also need a working Internet connection to download the source code and the external libraries required by some examples. Finally, you'll need a text editor to write your programs and access to a command shell to test and run them. We recommend using the Bash shell as most of the examples presented in the book use it.

For the most part, the example applications included in this book are compatible with any operating system supported by Go, such as Linux, macOS, and Windows. But when executing commands to interact with the operating system, for example, to create directories or list file contents, the book assumes you're running an operating system compatible with Linux or Unix standards. If you're using Windows, use the corresponding Windows commands to complete those tasks instead.

Typically, when the book instructs you to type commands, they'll look like this:

```
$ go test -v
```

The dollar sign ($) represents the shell prompt. You don't need to type it when typing the commands. Simply type the rest of the line after it.

3. golang.org/doc/install

Some examples present a series of commands and their output. In these cases, the lines starting with the dollar sign ($) represent the prompt and what you should type. The rest is the output from the commands:

```
$ ls
exceptionStep.go  main.go  main_test.go  step.go  testdata
$ go test -v
=== RUN   TestRun
=== RUN   TestRun/success
=== RUN   TestRun/fail
=== RUN   TestRun/failFormat
--- PASS: Test_Run (0.95s)
    --- PASS: TestRun/success (0.47s)
    --- PASS: TestRun/fail (0.03s)
    --- PASS: TestRun/failFormat (0.45s)
PASS
ok      pragprog.com/rggo/processes/goci      0.951s
```

In addition, throughout the book, you'll see that the examples are versioned from one section of the chapter to the next, by organizing the code into different directories, one for each version. This is necessary to make it easier to relate the examples with the source code shipped with the book. For example, in Chapter 5, Improving the Performance of Your CLI Tools, on page ?, the source code is split in the following directories:

```
$ find . -maxdepth 1 -mindepth 1 -type d | sort
./colStats
./colStats.v1
./colStats.v2
./colStats.v3
```

If you're developing the programs following the book, you don't need to create different directories for each version, unless you want to. In your case, it may be simpler to update the existing code according to the instructions in the book. You can also use a version control system, such as Git, to keep the different versions of your code available for reference.

If you've programmed with Go before, you'll notice that the code examples, as presented in the book, don't follow Go's formatting standards. In the book, the code is indented with spaces to ensure that the code fits the pages for printing. When copying the code from the book or downloading the examples, run gofmt to format the code automatically according to Go standards:

```
$ gofmt -w <source_file>.go
```

If your text editor is configured for automatic code reformatting, then save the file to update it according to the standards. Notice that there will be a difference between the code you see in the book's pages and the code you see in your text editor.

About the Example Code

The code provided in this book aims to illustrate concepts of the language that are useful when you're building your own programs and command-line tools.

This code isn't production-ready.

Even though each chapter shows a complete example you can use, the code may require additional features and checks to be used in real-life scenarios.

Go Modules

The code examples in this book rely on Go modules, the standard method to control and manage package dependencies for Go applications. By using Go modules, you can write your Go programs outside of the legacy $GOPATH required by older versions of Go prior to 1.11. Modules also enable reproducible builds as they record the specific version of Go and the external packages required to reliably build the application. You can find more information about Go modules in the official Go blog posts, *Using Go Modules*[4] and *New module changes in Go 1.16*.[5] To follow the examples, you need Go 1.13 or greater with modules enabled.

Modules provide a standard way to group related packages into a single unit that can be versioned together. They enable consistent dependency management for your Go applications. To use modules, create a directory for your code and use the go mod init command to initialize a new module, along with a unique module identifier. Typically, the unique module identifier is based on the version control path used to store the code. Since the code examples in the book aren't stored in a publicly accessible version control system, we'll use pragprog.com/rggo as the prefix for all example modules in the book. To better implement the book's examples, place your code in the $HOME/pragprog.com/rggo directory with sub-directories for each chapter. For example, initialize a new module like this:

```
$ mkdir -p $HOME/pragprog.com/rggo/firstProgram/wc
$ go mod init pragprog.com/rggo/firstProgram/wc
go: creating new go.mod: module pragprog.com/rggo/firstProgram/wc
```

4. blog.golang.org/using-go-modules
5. blog.golang.org/go116-module-changes

This directory is a suggestion. As long as you keep the same module path identifier when initializing the module, you can change the directory where you store the code, and you're still able to follow the examples as is.

Upon initializing a new module, Go creates a file go.mod in the root of your module directory and records the specific version of Go and the module path, like this:

```
$ cat go.mod
module pragprog.com/rggo/firstProgram/wc

go 1.16
```

If your code has external dependencies, you can record the specific version required in the go.mod file like this:

```
$ cat go.mod
module pragprog.com/rggo/workingFiles/mdp

go 1.16

require (
        github.com/microcosm-cc/bluemonday v1.0.2
        github.com/pmezard/go-difflib v1.0.0 // indirect
        github.com/russross/blackfriday/v2 v2.0.1
        github.com/shurcooL/sanitized_anchor_name v1.0.0 // indirect
)
```

For more details on how to use the external libraries with Go modules, consult their documentation.

Modules are fully integrated with the Go tooling. If you don't add the dependencies directly to your go.mod file, Go will automatically do that for you when you download it using go get. If you try to test or build your program using go test or go build, Go reports a missing dependency with a suggestion for how to obtain it. Go also creates a checksum file go.sum recording the specific checksum of each module used to build your program to ensure the next build uses the same version:

```
$ cat go.sum
github.com/microcosm-cc/bluemonday v1.0.2 h1:5lPfLTTAvAbtS0VqT+94yOtFnGfUWY...
github.com/microcosm-cc/bluemonday v1.0.2/go.mod h1:iVP4YcDBq+n/5fb23BhYFvI...
github.com/pmezard/go-difflib v1.0.0 h1:4DBwDE0NGyQoBHbLQYPwSUPoCMWR5BEzIk/...
github.com/pmezard/go-difflib v1.0.0/go.mod h1:iKH77koFhYxTK1pcRnkKkqfTogsb...
github.com/russross/blackfriday/v2 v2.0.1 h1:lPqVAte+HuHNfhJ/0LC98ESWRz8afy...
github.com/russross/blackfriday/v2 v2.0.1/go.mod h1:+Rmxgy9KzJVeS9/2gXHxylq...
github.com/shurcooL/sanitized_anchor_name v1.0.0 h1:PdmoCO6wvbs+7yrJyMORt4/...
github.com/shurcooL/sanitized_anchor_name v1.0.0/go.mod h1:1NzhyTcUVG4SuEtj...
golang.org/x/net v0.0.0-20181220203305-927f97764cc3 h1:eH6Eip3UpmR+yM/qI9Ij...
golang.org/x/net v0.0.0-20181220203305-927f97764cc3/go.mod h1:mL1N/T3taQHkD...
```

Notice that the output has been truncated to fit the book's page.

By using the files go.mod and go.sum, you ensure you're building the application with the same dependencies as the original developer. You can use these files provided with the book's source code to ensure your code will build exactly as shown in the book's examples.

Online Resources

The book's website[6] has links for downloading the source code and companion files.

If you're reading the electronic version of this book, you can click the box above the code excerpts to download that source code directly.

Now it's time to "go get" your feet wet. Let's start by developing a basic word counter command-line application that provides you with a working cross-platform tool. This also gives you an overview of the process for developing more complex applications.

6. https://pragprog.com/titles/rggo/