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

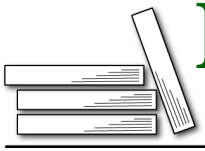
Powerful Command-Line Applications in Go

Build Fast and Maintainable Tools

Ricardo Gerardi

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

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: B1.0—October 23, 2019

Preface

Whether you're a software developer, a system administrators, a network specialists, or a DevOps engineer, with the advent of modern infrastructure technologies such as virtualisation, Software Defined Networks, containers, and the cloud, the number of devices and applications you have to manage has increased exponentially. It's becoming increasingly harder to deploy and maintain this infrastructure like you did in the past by depending on manual processes and static documentation.

Modern IT professionals are shifting towards using infrastructure as code to describe their environment and automation to deploy and manage them, relying on command line tools to ensure their IT and network infrastructures are running. These tools become critical components of the IT infrastructure and can no longer be treated as second-class citizens, requiring the same level of governance as other software components like version control, automated testing, and Continuous Integration/Continuous Deployment (CI/CD) practices. Many of these tools are written in teh Go programming language.

Go is a modern programming language that combines the reliability of compiled languages with the flexibility of dynamic typed languages. It is the ideal choice to write command line tools as it provides the easy of use and flexibility to quickly prototype new tools and ideas while also providing the features and tools required for more complex scenarios, including type safety, cross compilation, testing, benchmarks, and more, as they become necessary.

In this book, you'll leverage your knowledge of Go's basic syntax and apply more advanced concepts to develop different command line applications to automate tasks, analyze data, parse logs, talk to network services, or address many other systems requirements, while using tests and benchmarks 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'll start with the basic implementation, add some additional 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 to common input/output standards by applying different techniques. You'll take input from the Standard Input (STDIN) stream, as well as, parse command line parameters and define flags for your tool using the flags package. You will 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 work with files effectively in Go by developing 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 will 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 backup files according to different criteria. You'll perform common file system operations such as copying, compressing, and deleting files. You'll also log information onto the screen or to 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.

Then, 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 analyse its performance, find bottlenecks, and re-design 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 run tasks concurrently in a safe way.

the (as yet) unwritten Chapter 6, Controlling Processes, allows 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 to ensure your program does not run forever. You'll also handle operating system signals correctly allowing your tool to gracefully shut down.

In *the (as yet) unwritten Chapter 7, Talking to REST APIs,* , 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 test techniques that ensure your application works reliably without overloading web servers unnecessarily.

Next, in *the (as yet) unwritten Chapter 8, Using the Cobra CLI Framework,* , you'll develop a network tool that connects and executes tasks on remote machines via SSH, by leveraging the Cobra CLI Framework. Cobra is a popular framework that allows you to create flexible command line tools that use sub-commands compatible with the POSIX standard. You'll use Cobra to generate the boiler plate code for your application allowing you to focus on its business logic.

In *the (as yet) unwritten Chapter 9, Developing Interactive Terminal Tools,* , you'll build an interactive command line application that uses terminal widgets to interact with the user. You'll use Go's standard library and external packages to connect to standard databases by executing SQL operations. You will query, insert, and delete data from databases and use a local *Sqlite3* database to persist data for your tool.

In *the (as yet) unwritten Chapter 10, Working with Regular Expressions in Go,* , you'll use Regular Expressions with Go to find patterns of text in large text based datasets, including log files. You'll apply the `Regexp` type in different context ensuring pattern matching accuracy and good performance. You'll compile regular expressions to prevent errors. You'll then expand on these concepts to replace text with great flexibility.

And finally, in *the (as yet) unwritten Chapter 11, Distributing Your Tool,* , 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 behaviour of your builds according to external conditions. You'll take a quick look at "cgo" to embed C code in your Go applications. Then you'll apply techniques to package and distribute your application as a container, via Go Get, and using Ansible.

This book does not cover the basic syntax of the Go Program Language. You should be familiar with the basics such as declaring variables, types, custom types, flow control, and the general structure of a Go program. There are several other books and articles out there that do a great job explaining the language's syntax.

This book leverages 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 to create command line tools in general. By using the standard library, we benefit from Go's compatibility across different version making the code accessible to a larger number of readers. In some cases we'll use external packages when there's no equivalent functionality but, in general, we 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 *the (as yet) unwritten Chapter 8, Using the Cobra CLI Framework,* as it's a popular framework used by many developers and companies to extend Go's capabilities to manage command-line applications.

In each chapter, you will 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 for you to try in order to improve what you learned in the chapter and practice your skills further. And you're encouraged to add more features on your own.

The 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. There are two reasons for that: 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 in 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 seem more appealing, feel free to jump around. Keep in mind that some chapters build on skills presented in previous

chapters. There's usually a cross reference to where that concept was first discussed in the book so you can explore the topic in more details.

How to Use This Book

In order to follow this book and test the code examples, you need Go 1.11, or higher installed on your machine. At the time of writing this book, the current version of Go is 1.13. You can find more information on how to install Go in the official documentation [1].

The code examples in the book leverage *Go Modules* to manage package dependencies. For details, consult Go Modules, on page ix. By using Go Modules, you're no longer required to code under the `$GOPATH` directory. If you do not want to use modules or if you're using a version of Go older than 1.11, you can still follow most of the examples in the book, but you'll need to adjust the import paths and dependencies accordingly. This is out of scope of the book.

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

```
$ go test -v
```

1. golang.org/doc/install

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

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 required 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 instruction 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 programmed with Go before, you'll notice that the code examples, as presented in the book, do not 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 when you download the examples, run `gofmt` to automatically format the code 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 pages and the code you see in your text editor.

About 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 is not production ready.

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

Go Modules

In Go 1.11, the Go team released preliminary support for Go Modules, a new method to control and manage package dependencies for Go applications. With Go Modules you can write your Go programs outside of `$GOPATH`. Modules also enable reproducible builds as they record the specific version of Go and external packages required to reliably build the application. You can find more information about Go Modules in the official Go blog [2]. The code and examples in this book use Go Modules. If you want to follow the examples exactly as shown, you need Go 1.11 or later. Most of the examples should work with previous version of Go as long as you place all of your code in the `$GOPATH`, but you will need to modify the import path of packages to conform to a pre-modules standard. This is not covered in the book.

Modules provide a standard way to group related packages into a single unit that can be versioned together. Modules enable consistent dependency management for your Go applications. In order to use modules, create a directory for your code and use the `go mod init` command to initialize a new module, using a unique module identifier. Typically, the unique module identifier is based on the version control path where you store your code. Since the code examples in the book are not stored in a public accessible version control system, we'll use `pragprog.com/rggo` as the prefix to all example modules in the book. As a suggestion, the code will be stored in the `$HOME/pragprog.com/rggo` directory

2. blog.golang.org/using-go-modules

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
```

This directory is a suggestion. As long as you keep the same module path, 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 recording the specific version of Go and the module path, like this:

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

go 1.13
```

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 1.13

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 do not add the dependencies to your go.mod file, Go will do that for you automatically when you try to test or build your program using go test or go build. 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 exact 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 page.

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

Online Resources

The book's website[3] has links to download the source code and companion files for this book.

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, and an overview of the process to develop more complex applications.

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