

# THE POWER OF GO TOOLS

“Superb and well written”

—Lee Gibson



2024

BUILDING DELIGHTFUL SOFTWARE IN GO

JOHN ARUNDEL

# Contents

<b>Praise for ‘The Power of Go: Tools’</b>	<b>10</b>
<b>Introduction</b>	<b>11</b>
Who is this book for? . . . . .	11
What should I know before reading it? . . . . .	12
What version of Go does it cover? . . . . .	13
Where to find the code examples . . . . .	13
What you’ll learn . . . . .	13
<b>1. Packages</b>	<b>15</b>
The universal library . . . . .	15
Packages are a force multiplier . . . . .	16
The universal Go library is huge . . . . .	16
Sharing our code benefits us all . . . . .	17
Writing packages, not programs . . . . .	18
Command-line tools . . . . .	18
Zen mountaineering . . . . .	19
Guided by tests . . . . .	20
Building a hello package . . . . .	20
The structure of a test . . . . .	21
Tests are bug detectors . . . . .	22
So when should a test fail? . . . . .	22
Where does <code>fmt.Println</code> print to? . . . . .	23
Meet <code>bytes.Buffer</code> , an off-the-shelf <code>io.Writer</code> . . . . .	24
Failure standards . . . . .	25
Implementing hello, guided by tests . . . . .	25
We start with a failing test . . . . .	25
Creating a module . . . . .	26
One folder, one package . . . . .	26
A null implementation . . . . .	27
The real implementation . . . . .	27
The naming of tests . . . . .	28
Refactoring to use our new package . . . . .	28
Going further . . . . .	29
<b>2. Paperwork</b>	<b>30</b>
Making our package more flexible . . . . .	30
Mandatory arguments are annoying . . . . .	31
What if we allow users to pass <code>nil</code> ? . . . . .	31

Maybe some global variable instead? . . . . .	32
A struct is the answer . . . . .	33
A convenience wrapper with defaults . . . . .	34
A simple line counter . . . . .	36
Focus on behaviour, not implementation . . . . .	36
One possible first version . . . . .	37
What about configuration? . . . . .	39
Config structs don't solve the problem . . . . .	40
An elegant option API . . . . .	40
Okay, so what's an "option"? . . . . .	41
Options are functions . . . . .	42
"Always valid" fields . . . . .	43
Some internal paperwork . . . . .	45
Handling internal errors . . . . .	45
A line counter with options . . . . .	47
Methodical options . . . . .	50
Going further . . . . .	51
<b>3. Arguments</b>	<b>52</b>
Designing the behaviour . . . . .	53
Deciding the scope . . . . .	53
Testing CLI arguments . . . . .	53
A first attempt . . . . .	54
Test data files . . . . .	54
Creating the test data . . . . .	54
Using the data in a test . . . . .	55
The failing test . . . . .	55
Implementing file-reading . . . . .	56
Empty slice checking . . . . .	57
Testing the "no args" behaviour . . . . .	58
Closing files and other resources . . . . .	59
Updating the user interface . . . . .	59
Some user testing . . . . .	60
Setting exit status . . . . .	60
Test scripts . . . . .	62
Running the program as a binary . . . . .	62
Introducing testscript . . . . .	62
Invoking test scripts . . . . .	63
Defining custom commands . . . . .	64
The delegate Main function . . . . .	65
The stdout assertion . . . . .	65
Testing arguments . . . . .	66
Going further . . . . .	66
<b>4. Flags</b>	<b>68</b>
Commands . . . . .	68

The Unix way . . . . .	68
Multiple main packages . . . . .	69
A words command . . . . .	69
A test for word counting . . . . .	71
Updating the test scripts . . . . .	74
Flags . . . . .	76
Introducing flags . . . . .	76
Adding a <code>-lines</code> flag . . . . .	76
Implementing the behaviour . . . . .	79
Help and usage information . . . . .	81
Going further . . . . .	82
<b>5. Files</b>	<b>83</b>
Writing to files . . . . .	83
The art of judicious logging . . . . .	84
Testing a <code>WriteToFile</code> function . . . . .	84
Designing errors out of existence . . . . .	85
Looking for inspiration . . . . .	86
What are we really testing? . . . . .	86
The <code>go-cmp</code> module . . . . .	88
Implementing a <code>WriteToFile</code> function . . . . .	89
File permissions . . . . .	89
When the directory doesn't exist . . . . .	90
A disastrous bug . . . . .	90
Using <code>t.TempDir</code> . . . . .	92
Finishing the job . . . . .	93
It's clobbering time . . . . .	94
Ensuring permissions . . . . .	95
What's the worst that could happen? . . . . .	96
A security leak . . . . .	97
Going further . . . . .	99
<b>6. Filesystems</b>	<b>100</b>
Files and filesystems . . . . .	100
What even is a file? . . . . .	101
Organising files . . . . .	101
A simple file finder . . . . .	102
Handling folders recursively . . . . .	103
Filesystems and <code>io/fs</code> . . . . .	104
Matching files by name . . . . .	106
Walking the tree . . . . .	107
A file-finding tree-walker . . . . .	108
Starting at the top . . . . .	109
The <code>fs.FS</code> abstraction . . . . .	110
Any path-value map is a "filesystem" . . . . .	111
The <code>fstest.MapFS</code> filesystem . . . . .	111

Adding a filesystem to our API . . . . .	113
Timing potentially slow operations . . . . .	114
Writing benchmark functions . . . . .	115
Taking fs.FS makes APIs more flexible . . . . .	117
Going further . . . . .	119
<b>7. Commands</b>	<b>120</b>
The exec package . . . . .	120
What even is a process? . . . . .	120
Managing command output . . . . .	122
When not to use exec . . . . .	122
When to use exec . . . . .	124
Migrating from shell scripts to Go . . . . .	124
Why use Go to run commands? . . . . .	124
A command wrapper in Go . . . . .	125
The pmset command . . . . .	125
What can we test? . . . . .	125
Breaking down behaviour into chunks . . . . .	126
Parsing command output . . . . .	126
Testing the parsing function . . . . .	127
Parsing command output . . . . .	128
Clarifying the problem . . . . .	128
Writing a regular expression . . . . .	129
Using the regexp . . . . .	130
Integration tests . . . . .	131
Why isolate integration tests? . . . . .	132
Build tags . . . . .	132
Testing the command runner . . . . .	133
Running the command . . . . .	135
Capturing output . . . . .	135
When to import a third-party package . . . . .	136
Going further . . . . .	137
<b>8. Shells</b>	<b>139</b>
A simple shell . . . . .	139
Defining some behaviour . . . . .	140
Identifying the first test . . . . .	140
Comparing the incomparable . . . . .	141
Parsing user input . . . . .	142
Prototyping . . . . .	144
A pseudocode outline . . . . .	144
Main-driven development . . . . .	145
Feedback from user testing . . . . .	146
A stateful shell session . . . . .	148
What would we like to write? . . . . .	148
What object would make sense? . . . . .	149

Designing the Session object . . . . .	149
Running a session . . . . .	151
Testing the Run method . . . . .	151
Dependencies on external commands . . . . .	153
A dry-run mode . . . . .	153
Implementing Run . . . . .	154
What's still missing . . . . .	156
Globbing . . . . .	156
Redirection . . . . .	156
Piping . . . . .	156
Quoting . . . . .	157
Going further . . . . .	158
<b>9. Pipelines</b>	<b>159</b>
A realistic operations task . . . . .	159
Matching and counting log lines . . . . .	160
A quick shell spell . . . . .	160
Solving the problem with Go . . . . .	161
In what language would this be easy? . . . . .	163
Programs as pipelines . . . . .	163
A fluent API . . . . .	164
Errors in sequenced operations . . . . .	164
An error-safe writer . . . . .	166
Putting the pieces together . . . . .	167
How does data flow from one method to another? . . . . .	167
Testing the end of the pipeline . . . . .	167
Breaking ground . . . . .	169
Getting the test passing . . . . .	170
Adding error safety . . . . .	171
Obviousness-oriented programming . . . . .	172
Trying it out . . . . .	173
Hello, world . . . . .	173
The world strikes back . . . . .	174
Setting defaults with a constructor . . . . .	174
Reading data from files . . . . .	175
Another pipeline explosion . . . . .	176
Filtering data . . . . .	178
Extracting columns . . . . .	178
A String sink . . . . .	178
Testing Column . . . . .	180
Validating arguments . . . . .	181
Implementing Column . . . . .	182
The script package . . . . .	183
Was this a waste of time? . . . . .	184
Introducing script . . . . .	184
Some simple one-liners . . . . .	184

More sophisticated programs . . . . .	185
Concurrent pipeline stages . . . . .	186
Custom filter functions . . . . .	186
Solving problems . . . . .	187
The landscape of simple programs . . . . .	187
Going further . . . . .	188
<b>10. Data</b>	<b>189</b>
Marshalling . . . . .	189
Serialisation . . . . .	190
The gob package . . . . .	190
A file-based data store . . . . .	190
Testing data persistence . . . . .	191
Setters and getters . . . . .	191
A sensible key-value API . . . . .	191
Testing the key-value machinery . . . . .	192
Implementing the store . . . . .	194
Adding persistence . . . . .	195
What are we really testing here? . . . . .	195
An end-to-end persistence test . . . . .	196
Saving and loading . . . . .	197
Tightening up the tests . . . . .	198
JSON: a text data format . . . . .	200
The json package . . . . .	200
JSON is a useful auxiliary language . . . . .	200
A JSON output option . . . . .	201
Adding JSON to the battery package . . . . .	201
Producing JSON strings . . . . .	202
What could go wrong? . . . . .	202
Testing ToJSON . . . . .	203
Implementing ToJSON . . . . .	204
Pretty-printing with indentation . . . . .	205
Querying JSON output . . . . .	205
YAML: a less verbose JSON . . . . .	206
Parsing YAML in Go . . . . .	206
Designing the API with a test . . . . .	207
Defining our types . . . . .	208
The go-yaml package . . . . .	208
Decoding . . . . .	208
When unmarshalling doesn't work . . . . .	209
The format of struct tags . . . . .	210
Setting defaults . . . . .	210
Eliminating config structs . . . . .	211
Going further . . . . .	211
<b>11. Clients</b>	<b>212</b>

A simple weather client . . . . .	212
What do we need? . . . . .	213
Kicking the tyres . . . . .	213
Environmental credentials . . . . .	214
Making a GET request . . . . .	214
Initial user testing . . . . .	215
A second pass . . . . .	216
Parsing JSON responses . . . . .	217
Testing ParseResponse . . . . .	217
To decode or to unmarshal? . . . . .	218
A temporary struct type . . . . .	218
The response struct . . . . .	219
Implementing ParseResponse . . . . .	220
What could go wrong? . . . . .	220
Other kinds of invalid data . . . . .	221
What are we really testing here? . . . . .	222
Back to a running program . . . . .	222
Constructing the request URL . . . . .	223
A FormatURL function . . . . .	223
Getting the location as input . . . . .	224
Refactoring the remaining code . . . . .	225
A paperwork-reducing GetWeather function . . . . .	225
Testing against a local HTTP server . . . . .	225
A simple httptest example . . . . .	226
A trustful TLS client . . . . .	227
A weather client object . . . . .	227
A familiar pattern . . . . .	228
Refactoring the tests to use our client . . . . .	228
Writing the client constructor . . . . .	228
Refactoring GetWeather as a client method . . . . .	230
Testing GetWeather . . . . .	230
Implementing GetWeather . . . . .	231
A convenience wrapper . . . . .	231
Adding a Main function . . . . .	232
Adding temperature support . . . . .	234
Parsing temperature data . . . . .	234
More user testing . . . . .	235
Handling quantities with units . . . . .	235
Presenting temperatures in Celsius . . . . .	235
Defining a Temperature type . . . . .	236
Tackling more complex APIs . . . . .	237
Request data . . . . .	237
“CRUD” methods . . . . .	238
Last words . . . . .	238
Going further . . . . .	239



<b>About this book</b>	<b>240</b>
Who wrote this? . . . . .	240
Feedback . . . . .	240
Mailing list . . . . .	241
For the Love of Go . . . . .	241
The Power of Go: Tests . . . . .	242
Know Go: Generics . . . . .	243
Further reading . . . . .	245
Video course . . . . .	245
Credits . . . . .	246
<b>Acknowledgements</b>	<b>247</b>
<b>A sneak preview</b>	<b>248</b>
<b>1. Programming with confidence</b>	<b>249</b>
Self-testing code . . . . .	250
The adventure begins . . . . .	251
Verifying the test . . . . .	253
Running tests with <code>go test</code> . . . . .	254
Using <code>cmp.Diff</code> to compare results . . . . .	255
New behaviour? New test. . . . .	257
Test cases . . . . .	258
Adding cases one at a time . . . . .	260
Quelling a panic . . . . .	262
Refactoring . . . . .	263
Well, that was easy . . . . .	265
Sounds good, now what? . . . . .	266

# Praise for ‘The Power of Go: Tools’

*Curse you for derailing my day with another fascinating book! The content is absolutely awesome.*

—Peter Nunn

*Superb and well written: all the examples worked and were very helpful.*

—Lee Gibson

*It’s fantastic! I really like the approach.*

—Sal DiStefano

*The book does a great job of teaching what to do with Go. I really liked the emphasis on testing.*

—Pedro Sandoval

*Exactly the book I was looking for next! I love it.*

—Elliot Thomas

*What I really love about this book is it takes the newbie to the next level via real-world, relatable examples. The narrative flow clicks with me.*

—Rajaseelan Ganeswaran

*It’s clear, concise, and practical, especially for folks like me who enjoy the art of making end-user tools with Go. It’s really made me think when I write code about how simple and easy I can make it for folks to use, without, as John likes to say, ‘a lot of paperwork’.*

—Josh Feierman

*Everywhere I go, I’m asked if I think the university stifles writers. My opinion is that they don’t stifle enough of them.*

—Flannery O’Connor