

Vladimir Vivien, Mario Castro Contreras,
Mat Ryer

Go: Design Patterns for Real-World Projects

Learning Path

Build production-ready solutions in Go using cutting-edge technology and techniques



Packt

Go: Design Patterns for Real-World Projects

Build production-ready solutions in Go using
cutting-edge technology and techniques

A course in three modules



BIRMINGHAM - MUMBAI

Go: Design Patterns for Real-World Projects

Copyright © 2017 Packt Publishing

All rights reserved. No part of this course may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this course to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this course.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this course by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Published on: June, 2017

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78839-055-2

www.packtpub.com

Credits

Authors

Vladimir Vivien

Mario Castro Contreras

Mat Ryer

Content Development Editor

Nikhil Borkar

Graphics

Abhinash Sahu

Reviewers

Quintessence Anx

Parth Desai

Abhishek Kumar

Chris Schaefer

Shiju Varghese

Michael Hamrah

David Hernandez

Production Coordinator

Nilesh Mohite

Preface

The Go programming language has firmly established itself as a favorite for building complex and scalable system applications. Go offers a direct and practical approach to programming that lets programmers write correct and predictable code using concurrency idioms and a full-featured standard library.

What this learning path covers

Module 1, Learning Go programming, is a step-by-step, practical guide full of real world examples to help you get started with Go in no time at all. We start off by understanding the fundamentals of Go, followed by a detailed description of the Go data types, program structures and Maps. After this, you learn how to use Go concurrency idioms to avoid pitfalls and create programs that are exact in expected behavior. Next, you will be familiarized with the tools and libraries that are available in Go for writing and exercising tests, benchmarking, and code coverage.

Finally, you will be able to utilize some of the most important features of GO such as, Network Programming and OS integration to build efficient applications. All the concepts are explained in a crisp and concise manner and by the end of this module; you would be able to create highly efficient programs that you can deploy over cloud.

Module 2, Go Design Patterns, will provide readers with a reference point to software design patterns and CSP concurrency design patterns to help them build applications in a more idiomatic, robust, and convenient way in Go.

The module starts with a brief introduction to Go programming essentials and quickly moves on to explain the idea behind the creation of design patterns and how they appeared in the 90's as a common "language" between developers to solve common tasks in object-oriented programming languages. You will then learn how to apply the 23 Gang of Four (GoF) design patterns in Go and also learn about CSP concurrency patterns, the "killer feature" in Go that has helped Google develop software to maintain thousands of servers.

With all of this the module will enable you to understand and apply design patterns in an idiomatic way that will produce concise, readable, and maintainable software.

Module 3, Go Programming Blueprints - Second Edition, will show you how to leverage all the latest features and much more. This module shows you how to build powerful systems and drops you into real-world situations. You will learn to develop high-quality command-line tools that utilize the powerful shell capabilities and perform well using Go's in-built concurrency mechanisms. Scale, performance, and high availability lie at the heart of our projects, and the lessons learned throughout this module will arm you with everything you need to build world-class solutions. You will get a feel for app deployment using Docker and Google App Engine. Each project could form the basis of a start-up, which means they are directly applicable to modern software markets.

What you need for this learning path

Module 1:

To follow the examples in this module, you will need Go version 1.6 or later. Go supports architectures including AMD64, x386, and ARM running the following operating systems:

- Windows XP (or later)
- Mac OSX 10.7 (or later)
- Linux 2.6 (or later)
- FreeBSD 8 (or later)

Module 2:

Most of the chapters in this module are written following a simple TDD approach, here the requirements are written first, followed by some unit tests and finally the code that satisfies those requirements. We will use only tools that comes with the standard library of Go as a way to better understand the language and its possibilities. This idea is key to follow the module and understanding the way that Go solves problems, especially in distributed systems and concurrent applications.

Module 3:

To compile and run the code from this module, you will need a computer capable of running an operating system that supports the Go toolset, a list of which can be found at <https://golang.org/doc/install#requirements>

Appendix, Good Practices for a Stable Go Environment, has some useful tips to install Go and set up your development environment, including how to work with the GOPATH environment variable.

Who this learning path is for

Beginners to Go who are comfortable in other OOP languages like Java, C# or Python will find this course interesting and beneficial.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the course's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a course, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt course, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this course from your account at <http://www.packtpub.com>. If you purchased this course elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at <https://github.com/PacktPublishing/Go-Design-Patterns-for-Real-World-Projects>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our courses – maybe a mistake in the text or the code – we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your course, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the course in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this course, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Module 1: Learning Go Programming

Chapter 1: A First Step in Go	7
The Go programming language	8
Playing with Go	10
No IDE required	11
Installing Go	11
Source code examples	12
Your first Go program	12
Go in a nutshell	13
Functions	15
Packages	16
The workspace	16
Strongly typed	16
Composite types	17
The named type	18
Methods and objects	18
Interfaces	19
Concurrency and channels	20
Memory management and safety	21
Fast compilation	22
Testing and code coverage	22
Documentation	23
An extensive library	24
The Go Toolchain	25
Summary	25
Chapter 2: Go Language Essentials	26
The Go source file	27
Optional semicolon	29
Multiple lines	30
Go identifiers	32
The blank identifier	32
Muting package imports	32
Muting unwanted function results	33
Built-in identifiers	33

Types	33
Values	34
Functions	34
Go variables	34
Variable declaration	34
The zero-value	36
Initialized declaration	36
Omitting variable types	37
Short variable declaration	39
Restrictions for short variable declaration	40
Variable scope and visibility	40
Variable declaration block	42
Go constants	42
Constant literals	42
Typed constants	43
Untyped constants	43
Assigning untyped constants	44
Constant declaration block	45
Constant enumeration	46
Overriding the default enumeration type	47
Using iota in expressions	47
Skipping enumerated values	48
Go operators	49
Arithmetic operators	49
The increment and decrement operators	49
Go assignment operators	50
Bitwise operators	50
Logical Operators	51
Comparison operators	51
Operator precedence	52
Summary	52
Chapter 3: Go Control Flow	53
The if statement	53
The if statement initialization	57
Switch statements	57
Using expression switches	59
The fallthrough cases	60
Expressionless switches	62
Switch initializer	63

Type switches	64
The for statements	66
For condition	66
Infinite loop	67
The traditional for statement	68
The for range	70
The break, continue, and goto statements	73
The label identifier	73
The break statement	73
The continue statement	74
The goto statement	75
Summary	76
Chapter 4: Data Types	77
Go types	77
Numeric types	80
Unsigned integer types	81
Signed integer types	81
Floating point types	82
Complex number types	82
Numeric literals	82
Boolean type	84
Rune and string types	84
The rune	85
The string	86
Interpreted and raw string literals	87
Pointers	88
The pointer type	89
The address operator	89
The new() function	91
Pointer indirection - accessing referenced values	92
Type declaration	93
Type conversion	94
Summary	96
Chapter 5: Functions in Go	97
Go functions	97
Function declaration	98
The function type	101
Variadic parameters	102

Function result parameters	103
Named result parameters	104
Passing parameter values	105
Achieving pass-by-reference	106
Anonymous Functions and Closures	107
Invoking anonymous function literals	108
Closures	108
Higher-order functions	109
Error signaling and handling	110
Signaling errors	111
Error handling	114
The error type	114
Deferring function calls	115
Using defer	117
Function panic and recovery	117
Function panic	117
Function panic recovery	119
Summary	121
Chapter 6: Go Packages and Programs	122
The Go package	122
Understanding the Go package	122
The workspace	124
Creating a workspace	126
The import path	127
Creating packages	128
Declaring the package	129
Multi-File packages	130
Naming packages	131
Use globally unique namespaces	131
Add context to path	132
Use short names	132
Building packages	133
Installing a package	134
Package visibility	134
Package member visibility	135
Importing package	136
Specifying package identifiers	138
The dot identifier	139
The blank identifier	139

Package initialization	140
Creating programs	141
Accessing program arguments	143
Building and installing programs	145
Remote packages	146
Summary	147
Chapter 7: Composite Types	148
The array type	148
Array initialization	149
Declaring named array types	151
Using arrays	152
Array length and capacity	153
Array traversal	153
Array as parameters	154
The slice type	155
Slice initialization	156
Slice representation	157
Slicing	159
Slicing a slice	160
Slicing an array	161
Slice expressions with capacity	161
Making a slice	162
Using slices	163
Slices as parameters	164
Length and capacity	164
Appending to slices	165
Copying slices	165
Strings as slices	166
The map type	167
Map initialization	167
Making Maps	168
Using maps	169
Map traversal	170
Map functions	171
Maps as parameters	171
The struct type	172
Accessing struct fields	173
Struct initialization	173
Declaring named struct types	174

The anonymous field	175
Promoted fields	176
Structs as parameters	177
Field tags	178
Summary	179
Chapter 8: Methods, Interfaces, and Objects	180
Go methods	180
Value and pointer receivers	183
Objects in Go	185
The struct as object	186
Object composition	187
Field and method promotion	189
The constructor function	190
The interface type	191
Implementing an interface	192
Subtyping with Go interfaces	193
Implementing multiple interfaces	194
Interface embedding	196
The empty interface type	197
Type assertion	198
Summary	201
Chapter 9: Concurrency	202
Goroutines	202
The go statement	203
Goroutine scheduling	206
Channels	208
The Channel type	208
The send and receive operations	209
Unbuffered channel	209
Buffered channel	211
Unidirectional channels	212
Channel length and capacity	213
Closing a channel	214
Writing concurrent programs	215
Synchronization	215
Streaming data	217
Using for...range to receive data	219
Generator functions	220
Selecting from multiple channels	221

Channel timeout	223
The sync package	224
Synchronizing with mutex locks	224
Synchronizing access to composite values	226
Concurrency barriers with sync.WaitGroup	227
Detecting race conditions	228
Parallelism in Go	229
Summary	231
Chapter 10: Data IO in Go	232
IO with readers and writers	233
The io.Reader interface	233
Chaining readers	234
The io.Writer interface	236
Working with the io package	239
Working with files	241
Creating and opening files	241
Function os.OpenFile	242
Files writing and reading	243
Standard input, output, and error	245
Formatted IO with fmt	246
Printing to io.Writer interfaces	246
Printing to standard output	247
Reading from io.Reader	247
Reading from standard input	248
Buffered IO	249
Buffered writers and readers	249
Scanning the buffer	251
In-memory IO	252
Encoding and decoding data	253
Binary encoding with gob	254
Encoding data as JSON	256
Controlling JSON mapping with struct tags	259
Custom encoding and decoding	260
Summary	263
Chapter 11: Writing Networked Services	264
The net package	264
Addressing	264
The net.Conn Type	265

Dialing a connection	265
Listening for incoming connections	267
Accepting client connections	268
A TCP API server	269
Connecting to the TCP server with telnet	272
Connecting to the TCP server with Go	273
The HTTP package	275
The <code>http.Client</code> type	275
Configuring the client	277
Handling client requests and responses	278
A simple HTTP server	279
The default server	281
Routing requests with <code>http.ServeMux</code>	282
The default ServeMux	283
A JSON API server	284
Testing the API server with cURL	286
An API server client in Go	287
A JavaScript API server client	288
Summary	292
Chapter 12: Code Testing	293
The Go test tool	293
Test file names	294
Test organization	294
Writing Go tests	295
The test functions	296
Running the tests	298
Filtering executed tests	299
Test logging	300
Reporting failure	301
Skipping tests	302
Table-driven tests	304
HTTP testing	305
Testing HTTP server code	306
Testing HTTP client code	307
Test coverage	310
The <code>cover</code> tool	310
Code benchmark	312
Running the benchmark	313
Skipping test functions	313

The benchmark report	314
Adjusting N	314
Comparative benchmarks	315
Summary	317
Index	318

Module 2: Go Design Patterns

Chapter 1: Ready... Steady... Go!	8
A little bit of history	9
Installing Go	9
Linux	10
Go Linux advanced installation	10
Windows	11
Mac OS X	11
Setting the workspace - Linux and Apple OS X	12
Starting with Hello World	13
Integrated Development Environment - IDE	14
Types	15
Variables and constants	16
Operators	17
Flow control	18
The if... else statement	18
The switch statement	19
The for...range statement	19
Functions	20
What does a function look like?	20
What is an anonymous function?	21
Closures	22
Creating errors, handling errors and returning errors.	22
Function with undetermined number of parameters	23
Naming returned types	24
Arrays, slices, and maps	24
Arrays	24
Zero-initialization	24
Slices	25
Maps	26
Visibility	26
Zero-initialization	27
Pointers and structures	29
What is a pointer? Why are they good?	29
Structs	30

Interfaces	32
Interfaces - signing a contract	32
Testing and TDD	34
The testing package	35
What is TDD?	37
Libraries	38
The Go get tool	41
Managing JSON data	42
The encoding package	43
Go tools	45
The golint tool	45
The gofmt tool	46
The godoc tool	47
The goimport tool	47
Contributing to Go open source projects in GitHub	48
Summary	49
Chapter 2: Creational Patterns - Singleton, Builder, Factory, Prototype, and Abstract Factory Design Patterns	50
Singleton design pattern - having a unique instance of a type in the entire program	50
Description	50
Objectives	51
Example - a unique counter	51
Requirements and acceptance criteria	52
Writing unit tests first	52
Implementation	54
A few words about the Singleton design pattern	56
Builder design pattern - reusing an algorithm to create many implementations of an interface	56
Description	57
Objectives	57
Example - vehicle manufacturing	57
Requirements and acceptance criteria	58
Unit test for the vehicle builder	58
Implementation	62
Wrapping up the Builder design pattern	65
Factory method - delegating the creation of different types of payments	66
Description	66
Objectives	66

The example - a factory of payment methods for a shop	67
Acceptance criteria	67
First unit test	67
Implementation	70
Upgrading the Debitcard method to a new platform	73
What we learned about the Factory method	74
Abstract Factory - a factory of factories	75
Description	75
The objectives	75
The vehicle factory example, again?	76
Acceptance criteria	76
Unit test	76
Implementation	82
A few lines about the Abstract Factory method	83
Prototype design pattern	84
Description	84
Objective	84
Example	84
Acceptance criteria	85
Unit test	85
Implementation	88
What we learned about the Prototype design pattern	90
Summary	90
Chapter 3: Structural Patterns - Composite, Adapter, and Bridge Design Patterns	91
Composite design pattern	91
Description	92
Objectives	92
The swimmer and the fish	93
Requirements and acceptance criteria	93
Creating compositions	93
Binary Tree compositions	97
Composite pattern versus inheritance	98
Final words on the Composite pattern	99
Adapter design pattern	99
Description	99
Objectives	100
Using an incompatible interface with an Adapter object	100
Requirements and acceptance criteria	100

Unit testing our Printer adapter	100
Implementation	103
Examples of the Adapter pattern in Go's source code	104
What the Go source code tells us about the Adapter pattern	108
Bridge design pattern	108
Description	109
Objectives	109
Two printers and two ways of printing for each	109
Requirements and acceptance criteria	109
Unit testing the Bridge pattern	110
Implementation	117
Reuse everything with the Bridge pattern	120
Summary	121
Chapter 4: Structural Patterns - Proxy, Facade, Decorator, and Flyweight Design Patterns	122
Proxy design pattern	122
Description	122
Objectives	123
Example	123
Acceptance criteria	123
Unit test	123
Implementation	128
Proxying around actions	131
Decorator design pattern	131
Description	131
Objectives	132
Example	132
Acceptance criteria	132
Unit test	133
Implementation	137
A real-life example - server middleware	139
Starting with the common interface, http.Handler	140
A few words about Go's structural typing	145
Summarizing the Decorator design pattern - Proxy versus Decorator	146
Facade design pattern	146
Description	146
Objectives	147
Example	147
Acceptance criteria	147

Unit test	148
Implementation	152
Library created with the Facade pattern	154
Flyweight design pattern	155
Description	155
Objectives	155
Example	156
Acceptance criteria	156
Basic structs and tests	156
Implementation	159
What's the difference between Singleton and Flyweight then?	163
Summary	164
Chapter 5: Behavioral Patterns - Strategy, Chain of Responsibility, and Command Design Patterns	165
Strategy design pattern	165
Description	166
Objectives	166
Rendering images or text	166
Acceptance criteria	167
Implementation	168
Solving small issues in our library	173
Final words on the Strategy pattern	180
Chain of responsibility design pattern	180
Description	181
Objectives	181
A multi-logger chain	181
Unit test	182
Implementation	187
What about a closure?	190
Putting it together	192
Command design pattern	192
Description	192
Objectives	193
A simple queue	194
Acceptance criteria	194
Implementation	195
More examples	197
Chain of responsibility of commands	199
Rounding-up the Command pattern up	201

Summary	201
Chapter 6: Behavioral Patterns - Template, Memento, and Interpreter Design Patterns	203
Template design pattern	203
Description	204
Objectives	204
Example - a simple algorithm with a deferred step	204
Requirements and acceptance criteria	205
Unit tests for the simple algorithm	205
Implementing the Template pattern	207
Anonymous functions	208
How to avoid modifications on the interface	211
Looking for the Template pattern in Go's source code	214
Summarizing the Template design pattern	215
Memento design pattern	216
Description	216
Objectives	216
A simple example with strings	217
Requirements and acceptance criteria	217
Unit test	217
Implementing the Memento pattern	221
Another example using the Command and Facade patterns	223
Last words on the Memento pattern	227
Interpreter design pattern	227
Description	228
Objectives	228
Example - a polish notation calculator	228
Acceptance criteria for the calculator	228
Unit test of some operations	229
Implementation	230
Complexity with the Interpreter design pattern	234
Interpreter pattern again - now using interfaces	235
The power of the Interpreter pattern	238
Summary	238
Chapter 7: Behavioral Patterns - Visitor, State, Mediator, and Observer Design Patterns	239
Visitor design pattern	239
Description	240

Objectives	240
A log appender	240
Acceptance criteria	241
Unit tests	241
Implementation of Visitor pattern	245
Another example	247
Visitors to the rescue!	252
State design pattern	252
Description	252
Objectives	252
A small guess the number game	253
Acceptance criteria	253
Implementation of State pattern	253
A state to win and a state to lose	258
The game built using the State pattern	259
Mediator design pattern	259
Description	259
Objectives	259
A calculator	260
Acceptance criteria	260
Implementation	260
Uncoupling two types with the Mediator	264
Observer design pattern	264
Description	264
Objectives	265
The notifier	265
Acceptance criteria	265
Unit tests	266
Implementation	270
Summary	274
Chapter 8: Introduction to Gos Concurrency	275
A little bit of history and theory	275
Concurrency versus parallelism	276
CSP versus actor-based concurrency	278
Goroutines	279
Our first Goroutine	279
Anonymous functions launched as new Goroutines	281
WaitGroups	282
Callbacks	285

Callback hell	287
Mutexes	288
An example with mutexes - concurrent counter	289
Presenting the race detector	290
Channels	293
Our first channel	293
Buffered channels	295
Directional channels	297
The select statement	298
Ranging over channels too!	302
Using it all - concurrent singleton	303
Unit test	303
Implementation	304
Summary	309
Chapter 9: Concurrency Patterns - Barrier, Future, and Pipeline Design Patterns	310
 Barrier concurrency pattern	311
Description	311
Objectives	311
An HTTP GET aggregator	311
Acceptance criteria	312
Unit test - integration	313
Implementation	316
Waiting for responses with the Barrier design pattern	320
 Future design pattern	321
Description	321
Objectives	323
A simple asynchronous requester	323
Acceptance criteria	323
Unit tests	324
Implementation	328
Putting the Future together	332
 Pipeline design pattern	332
Description	332
Objectives	332
A concurrent multi-operation	333
Acceptance criteria	333
Beginning with tests	333
Implementation	335

The list generator	338
Raising numbers to the power of 2	338
Final reduce operation	339
Launching the Pipeline pattern	339
Final words on the Pipeline pattern	340
Summary	340
Chapter 10: Concurrency Patterns - Workers Pool and Publish/Subscriber Design Patterns	342
Workers pool	343
Description	343
Objectives	343
A pool of pipelines	343
Acceptance criteria	344
Implementation	344
The dispatcher	345
The pipeline	348
An app using the workers pool	351
No tests?	353
Wrapping up the Worker pool	355
Concurrent Publish/Subscriber design pattern	356
Description	356
Objectives	356
Example - a concurrent notifier	357
Acceptance criteria	358
Unit test	359
Testing subscriber	359
Testing publisher	363
Implementation	367
Implementing the publisher	370
Handling channels without race conditions	371
A few words on the concurrent Observer pattern	373
Summary	374
Index	375

Module 3: Go Programming Blueprints, Second Edition

Chapter 1: Chat Application with Web Sockets	9
A simple web server	10
Separating views from logic using templates	12
Doing things once	14
Using your own handlers	14
Properly building and executing Go programs	15
Modeling a chat room and clients on the server	15
Modeling the client	16
Modeling a room	19
Concurrency programming using idiomatic Go	19
Turning a room into an HTTP handler	20
Using helper functions to remove complexity	22
Creating and using rooms	23
Building an HTML and JavaScript chat client	23
Getting more out of templates	25
Tracing code to get a look under the hood	28
Writing a package using TDD	28
Interfaces	29
Unit tests	30
Red-green testing	32
Implementing the interface	34
Unexported types being returned to users	35
Using our new trace package	36
Making tracing optional	38
Clean package APIs	39
Summary	40
Chapter 2: Adding User Accounts	41
Handlers all the way down	42
Making a pretty social sign-in page	45
Endpoints with dynamic paths	47
Getting started with OAuth2	50
Open source OAuth2 packages	50
Tell the authorization providers about your app	51

Implementing external logging in	52
Logging in	53
Handling the response from the provider	56
Presenting the user data	58
Augmenting messages with additional data	59
Summary	64
Chapter 3: Three Ways to Implement Profile Pictures	65
Avatars from the OAuth2 server	66
Getting the avatar URL	66
Transmitting the avatar URL	67
Adding the avatar to the user interface	68
Logging out	69
Making things prettier	71
Implementing Gravatar	73
Abstracting the avatar URL process	73
The auth service and the avatar's implementation	74
Using an implementation	76
The Gravatar implementation	78
Uploading an avatar picture	81
User identification	82
An upload form	83
Handling the upload	84
Serving the images	86
The Avatar implementation for local files	87
Supporting different file types	89
Refactoring and optimizing our code	90
Replacing concrete types with interfaces	91
Changing interfaces in a test-driven way	92
Fixing the existing implementations	94
Global variables versus fields	95
Implementing our new design	96
Tidying up and testing	97
Combining all three implementations	98
Summary	100
Chapter 4: Command-Line Tools to Find Domain Names	101
Pipe design for command-line tools	102
Five simple programs	102
Sprinkle	103
Domainify	107
Coolify	109

Synonyms	112
Using environment variables for configuration	113
Consuming a web API	113
Getting domain suggestions	117
Available	118
Composing all five programs	122
One program to rule them all	123
Summary	127
Chapter 5: Building Distributed Systems and Working with Flexible Data	128
The system design	129
The database design	130
Installing the environment	131
Introducing NSQ	131
NSQ driver for Go	133
Introducing MongoDB	133
MongoDB driver for Go	134
Starting the environment	134
Reading votes from Twitter	135
Authorization with Twitter	135
Extracting the connection	137
Reading environment variables	138
Reading from MongoDB	140
Reading from Twitter	142
Signal channels	144
Publishing to NSQ	146
Gracefully starting and stopping programs	148
Testing	150
Counting votes	151
Connecting to the database	152
Consuming messages in NSQ	153
Keeping the database updated	155
Responding to Ctrl + C	157
Running our solution	158
Summary	159
Chapter 6: Exposing Data and Functionality through a RESTful Data Web Service API	161
RESTful API design	162
Sharing data between handlers	163
Context keys	163

Wrapping handler functions	165
API keys	165
Cross-origin resource sharing	166
Injecting dependencies	167
Responding	167
Understanding the request	169
Serving our API with one function	171
Using handler function wrappers	173
Handling endpoints	173
Using tags to add metadata to structs	174
Many operations with a single handler	174
Reading polls	175
Creating a poll	178
Deleting a poll	179
CORS support	180
Testing our API using curl	180
A web client that consumes the API	182
Index page showing a list of polls	183
Creating a new poll	185
Showing the details of a poll	186
Running the solution	189
Summary	191
Chapter 7: Random Recommendations Web Service	193
<hr/>	
The project overview	194
Project design specifics	195
Representing data in code	197
Public views of Go structs	200
Generating random recommendations	201
The Google Places API key	203
Enumerators in Go	203
Test-driven enumerator	205
Querying the Google Places API	209
Building recommendations	210
Handlers that use query parameters	212
CORS	213
Testing our API	214
Web application	216
Summary	216
Chapter 8: Filesystem Backup	218

Solution design	219
The project structure	219
The backup package	220
Considering obvious interfaces first	220
Testing interfaces by implementing them	221
Has the filesystem changed?	224
Checking for changes and initiating a backup	226
Hardcoding is OK for a short while	228
The user command-line tool	229
Persisting small data	230
Parsing arguments	231
Listing the paths	232
String representations for your own types	232
Adding paths	233
Removing paths	233
Using our new tool	234
The daemon backup tool	235
Duplicated structures	237
Caching data	237
Infinite loops	238
Updating filedb records	239
Testing our solution	240
Summary	242
Chapter 9: Building a Q&A Application for Google App Engine	243
The Google App Engine SDK for Go	244
Creating your application	245
App Engine applications are Go packages	246
The app.yaml file	246
Running simple applications locally	247
Deploying simple applications to Google App Engine	249
Modules in Google App Engine	250
Specifying modules	251
Routing to modules with dispatch.yaml	252
Google Cloud Datastore	252
Denormalizing data	253
Entities and data access	255
Keys in Google Cloud Datastore	256
Putting data into Google Cloud Datastore	257
Reading data from Google Cloud Datastore	259
Google App Engine users	259

Embedding denormalized data	261
Transactions in Google Cloud Datastore	
Using transactions to maintain counters	263
Avoiding early abstraction	267
Querying in Google Cloud Datastore	
Votes	
Indexing	270
Embedding a different view of entities	271
Casting a vote	
Accessing parents via datastore.Key	274
Line of sight in code	274
Exposing data operations over HTTP	
Optional features with type assertions	277
Response helpers	278
Parsing path parameters	279
Exposing functionality via an HTTP API	281
HTTP routing in Go	281
Context in Google App Engine	282
Decoding key strings	283
Using query parameters	285
Anonymous structs for request data	286
Writing self-similar code	287
Validation methods that return an error	288
Mapping the router handlers	289
Running apps with multiple modules	
Testing locally	290
Using the admin console	291
Automatically generated indexes	292
Deploying apps with multiple modules	
Summary	293
Chapter 10: Micro-services in Go with the Go kit Framework	294
Introducing gRPC	296
Protocol buffers	297
Installing protocol buffers	298
Protocol buffers language	298
Generating Go code	300
Building the service	301
Starting with tests	302
Constructors in Go	303
Hashing and validating passwords with bcrypt	304

Modeling method calls with requests and responses	305
Endpoints in Go kit	307
Making endpoints for service methods	308
Different levels of error	309
Wrapping endpoints into a Service implementation	309
An HTTP server in Go kit	311
A gRPC server in Go kit	312
Translating from protocol buffer types to our types	313
Creating a server command	315
Using Go kit endpoints	318
Running the HTTP server	318
Running the gRPC server	319
Preventing a main function from terminating immediately	320
Consuming the service over HTTP	320
Building a gRPC client	321
A command-line tool to consume the service	323
Parsing arguments in CLIs	324
Maintaining good line of sight by extracting case bodies	325
Installing tools from the Go source code	326
Rate limiting with service middleware	327
Middleware in Go kit	328
Manually testing the rate limiter	330
Graceful rate limiting	331
Summary	332
Chapter 11: Deploying Go Applications Using Docker	333
Using Docker locally	334
Installing Docker tools	334
Dockerfile	334
Building Go binaries for different architectures	335
Building a Docker image	336
Running a Docker image locally	337
Inspecting Docker processes	338
Stopping a Docker instance	339
Deploying Docker images	339
Deploying to Docker Hub	339
Deploying to Digital Ocean	341
Creating a droplet	341
Accessing the droplet's console	344
Pulling Docker images	346

Running Docker images in the cloud	348
Accessing Docker images in the cloud	348
Summary	349
Chapter 12: Good Practices for a Stable Go Environment	350
Installing Go	350
Configuring Go	351
Getting GOPATH right	352
Go tools	353
Cleaning up, building, and running tests on save	355
Integrated developer environments	356
Sublime Text 3	356
Visual Studio Code	359
Summary	362
Index	363

Module 1

Learning Go Programming

An insightful guide to learning the Go programming language

1

A First Step in Go

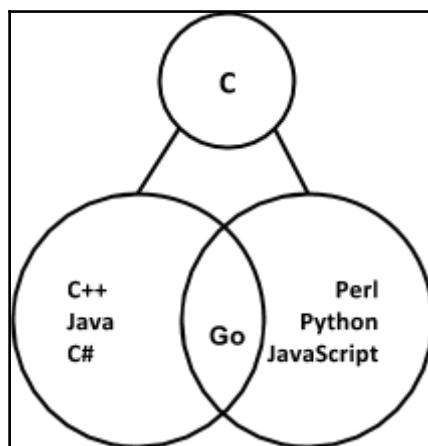
In the first chapter of the book, you will be introduced to Go and take a tour of the features that have made the language a favorite among its adopters. The start of the chapter provides the motivation behind the Go programming language. If you are impatient, however, you are welcome to skip to any of the other topics and learn how to write your first Go program. Finally, the *Go in a nutshell* section provides a high-level summary of the characteristics of the language.

The following topics are covered in this chapter:

- The Go programming language
- Playing with Go
- Installing Go
- Your first Go program
- Go in a nutshell

The Go programming language

Since the invention of the C language in the early 1970s by *Dennis Ritchie* at Bell Labs, the computing industry has produced many popular languages that are based directly on (or have borrowed ideas from) its syntax. Commonly known as the C-family of languages, they can be split into two broad evolutionary branches. In one branch, derivatives such as C++, C#, and Java have evolved to adopt a strong type system, object orientation, and the use of compiled binaries. These languages, however, tend to have a slow build-deploy cycle and programmers are forced to adopt a complex object-oriented type system to attain runtime safety and speed of execution:



In the other evolutionary linguistic branch are languages such as Perl, Python, and JavaScript that are described as dynamic languages for their lack of type safety formalities, use of lightweight scripting syntax, and code interpretation instead of compilation. Dynamic languages have become the preferred tool for web and cloud scale development where speed and ease of deployment are valued over runtime safety. The interpreted nature of dynamic languages means, however, they generally run slower than their compiled counterparts. In addition, the lack of type safety at runtime means the correctness of the system scales poorly as the application grows.

Go was created as a system language at Google in 2007 by *Robert Griesemer, Rob Pike, and Ken Thomson* to handle the needs of application development. The designers of Go wanted to mitigate the issues with the aforementioned languages while creating a new language that is simple, safe, consistent, and predictable. As Rob Pike puts it:

"Go is an attempt to combine the safety and performance of a statically-typed language with the expressiveness and convenience of a dynamically-typed interpreted language."

Go borrows ideas from different languages that came before it, including:

- Simplified but concise syntax that is fun and easy to use
- A type of system that feels more like a dynamic language
- Support for object-oriented programming
- Statically typed for compilation and runtime safety
- Compiled to native binaries for fast runtime execution
- Near-zero compilation time that feels more like an interpreted language
- A simple concurrency idiom to leverage multi-core, multi-chip machines
- A garbage collector for safe and automatic memory management

The remainder of this chapter will walk you through an introductory set of steps that will give you a preview of the language and get you started with building and running your first Go program. It is a precursor to the topics that are covered in detail in the remaining chapters of the book. You are welcome to skip to other chapters if you already have a basic understanding of Go.

Playing with Go

Before we jump head-first into installing and running Go tools on your local machine, let us take a look at the **Go Playground**. The creators of the language have made available a simple way to familiarize yourself with the language without installing any tools. Known as the Go Playground, it is a web-based tool, accessible from <https://play.golang.org/>, that uses an editor metaphor to let developers test their Go skills by writing code directly within the web browser window. The Playground gives its users the ability to compile and run their code on Google's remote servers and get immediate results as shown in the following screenshot:

The screenshot shows a web browser window titled "The Go Playground". The address bar displays the URL <https://play.golang.org/>. The main content area is titled "The Go Playground" and contains a code editor with the following Go code:

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     fmt.Println("Hello, playground")
9 }
10
11
12
13
14
15
```

Below the code editor, the output of the program is displayed:

```
Hello, playground
Program exited.
```

The editor is basic, as it is meant to be used as a learning tool and a way to share code with others. The Playground includes practical features such as line numbers and formatting to ensure your code remains readable as it goes beyond a few lines long. Since this is a free service that consumes real compute resources, Google understandably imposes a few limitations on what can be done with Playground:

- You are restricted on the amount of memory your code will consume
- Long-running programs will be killed

- Access to files is simulated with an in-memory filesystem.
- Network access is simulated against the loopback interface only

No IDE required

Besides the Go Playground, how is one supposed to write Go code anyway? Writing Go does not require a fancy **Integrated Development Environment (IDE)**. As a matter of fact, you can get started writing your simple Go programs with your favorite plain text editor that is bundled with your OS. There are, however, Go plugins for most major text editors (and full-blown IDEs) such as Atom, Vim, Emacs, Microsoft Code, IntelliJ, and many others. There is a complete list of editors and IDE plugins for Go which can be found at <https://github.com/golang/go/wiki/IDEsAndTextEditorPlugins>.

Installing Go

To start programming with Go on your local machine you will need to install the **Go Toolchain** on your computer. At the time of writing, Go comes ready to be installed on the following major OS platforms:

- Linux
- FreeBSD Unix
- Mac OSX
- Windows

The official installation packages are all available for 32-bit and 64-bit Intel-based architectures. There are also official binary releases that are available for ARM architectures as well. As Go grows in popularity, there will certainly be more binary distribution choices made available in the future.

Let us skip the detailed installation instructions as they will certainly change by the time you read this. Instead, you are invited to visit <http://golang.org/doc/install> and follow the directions given for your specific platform. Once completed, be sure to test your installation is working before continuing to use the following command:

```
$> go version  
go version go1.6.1 linux/amd64
```

The previous command should print the version number, target OS, and the machine architecture where Go and its tools are installed. If you do not get an output similar to that preceding command, ensure to add the path of the Go binaries to your OS's execution PATH environment variable.

Before you start writing your own code, ensure that you have properly set up your GOPATH. This is a local directory where your Go source files and compiled artifacts are saved as you use the Go Toolchain. Follow the instructions found in <https://golang.org/doc/install#testing> to set up your GOPATH.

Source code examples

The programming examples presented throughout this book are available on the GitHub source code repository service. There you will find all source files grouped by chapters in the repository at <https://github.com/vladimirvivien/learning-go/>. To save the readers a few keystrokes, the examples use a shortened URL, that starts with `golang.fyi`, that points directly to the respective file in GitHub.

Alternatively, you can follow along by downloading and unzipping (or cloning) the repository locally. Create a directory structure in your GOPATH so that the root of the source files is located at `$GOPATH/src/github.com/vladimirvivien/learning-go/`.

Your first Go program

After installing the Go tools successfully on your local machine, you are now ready to write and execute your first Go program. For that, simply open your favorite text editor and type in the simple Hello World program shown in the following code:

```
package main
import "fmt"
func main() {
    fmt.Println("Hello, World!")
}
```

`golang.fyi/ch01/helloworld.go`

Save the source code in a file called `helloworld.go` anywhere inside your GOPATH. Then use the following Go command to compile and run the program:

```
$> go run helloworld.go
Hello, World!
```

If all goes well, you should see the message **Hello, World!** output on your screen. Congratulations, you have just written and executed your first Go program. Now, let us explore the attributes and characteristics of the Go language at a high level.

Go in a nutshell

By design, Go has a simple syntax. Its designers wanted to create a language that is clear, concise, and consistent with few syntactic surprises. When reading Go code, keep this mantra in mind: *what you see is what it is*. Go shies away from a clever and terse coding style in favor of code that is clear and readable as exemplified by the following program:

```
// This program prints molecular information for known metalloids
// including atomic number, mass, and atom count found
// in 100 grams of each element using the mole unit.
// See http://en.wikipedia.org/wiki/Mole\_\(unit\)
package main

import "fmt"

const avogadro float64 = 6.0221413e+23
const grams = 100.0

type amu float64

func (mass amu) float() float64 {
    return float64(mass)
}

type metalloid struct {
    name    string
    number int32
    weight amu
}

var metalloids = []metalloid{
    metalloid{"Boron", 5, 10.81},
    metalloid{"Silicon", 14, 28.085},
    metalloid{"Germanium", 32, 74.63},
    metalloid{"Arsenic", 33, 74.921},
```

```
metalloid{"Antimony", 51, 121.760},
metalloid{"Tellerium", 52, 127.60},
metalloid{"Polonium", 84, 209.0},
}

// finds # of moles
func moles(mass amu) float64 {
return grams / float64(mass)
}

// returns # of atoms moles
func atoms(moles float64) float64 {
    return moles * avogadro
}

// return column headers
func headers() string {
    return fmt.Sprintf(
        "%-10s %-10s %-10s Atoms in %.2f Grams\n",
        "Element", "Number", "AMU", grams,
    )
}

func main() {
    fmt.Println(headers())

    for _, m := range metalloids {
        fmt.Printf(
            "%-10s %-10d %-10.3f %e\n",
            m.name, m.number, m.weight.float(), atoms(moles(m.weight)),
        )
    }
}
```

golang.fyi/ch01/metalloids.go

When the code is executed, it will give the following output:

```
$> go run metalloids.go
Element      Number      AMU      Atoms in 100.00 Grams
Boron        5          10.810    6.509935e+22
Silicon      14         28.085    1.691318e+23
Germanium    32         74.630    4.494324e+23
Arsenic      33         74.921    4.511848e+23
Antimony     51         121.760   7.332559e+23
Tellerium    52         127.600   7.684252e+23
Polonium     84         209.000   1.258628e+24
```

If you have never seen Go before, you may not understand some of the details of the syntax and idioms used in the previous program. Nevertheless, when you read the code, there is a good chance you will be able to follow the logic and form a mental model of the program's flow. That is the beauty of Go's simplicity and the reason why so many programmers use it. If you are completely lost, no need to worry, as the subsequent chapters will cover all aspects of the language to get you going.

Functions

Go programs are composed of functions, the smallest callable code unit in the language. In Go, functions are typed entities that can either be named (as shown in the previous example) or be assigned to a variable as a value:

```
// a simple Go function
func moles(mass amu) float64 {
    return float64(mass) / grams
}
```

Another interesting feature about Go functions is their ability to return multiple values as a result of a call. For instance, the previous function could be re-written to return a value of type `error` in addition to the calculated `float64` value:

```
func moles(mass amu) (float64, error) {
    if mass < 0 {
        return 0, error.New("invalid mass")
    }
    return (float64(mass) / grams), nil
}
```

The previous code uses the multi-return capabilities of Go functions to return both the mass and an error value. You will encounter this idiom throughout the book used as a mean to properly signal errors to the caller of a function. There will be further discussion on multi-return value functions covered in Chapter 5, *Functions in Go*.

Packages

Source files containing Go functions can be further organized into directory structures known as a package. Packages are logical modules that are used to share code in Go as libraries. You can create your own local packages or use tools provided by Go to automatically pull and use remote packages from a source code repository. You will learn more about Go packages in Chapter 6, *Go Packages and Programs*.

The workspace

Go follows a simple code layout convention to reliably organize source code packages and to manage their dependencies. Your local Go source code is stored in the workspace, which is a directory convention that contains the source code and runtime artifacts. This makes it easy for Go tools to automatically find, build, and install compiled binaries. Additionally, Go tools rely on the workspace setup to pull source code packages from remote repositories, such as Git, Mercurial, and Subversion, and satisfy their dependencies.

Strongly typed

All values in Go are statically typed. However, the language offers a simple but expressive type system that can have the feel of a dynamic language. For instance, types can be safely inferred as shown in the following code snippet:

```
const grams = 100.0
```

As you would expect, constant grams would be assigned a numeric type, `float64`, to be precise, by the Go type system. This is true not only for constants, but any variable can use a short-hand form of declaration and assignment as shown in the following example:

```
package main
import "fmt"
func main() {
    var name = "Metalloids"
    var triple = [3]int{5,14,84}
    elements := []string{"Boron", "Silicon", "Polonium"}
    isMetal := false
    fmt.Println(name, triple, elements, isMetal)
}
```

Notice that the variables, in the previous code snippet, are not explicitly assigned a type. Instead, the type system assigns each variable a type based on the literal value in the assignment. Chapter 2, *Go Language Essentials* and Chapter 4, *Data Types*, go into more details regarding Go types.

Composite types

Besides the types for simple values, Go also supports composite types such as `array`, `slice`, and `map`. These types are designed to store indexed elements of values of a specified type. For instance, the `metalloid` example shown previously makes use of a slice, which is a variable-sized array. The variable `metalloid` is declared as a `slice` to store a collection of the type `metalloid`. The code uses the literal syntax to combine the declaration and assignment of a slice of type `metalloid`:

```
var metalloids = []metalloid{
    metalloid{"Boron", 5, 10.81},
    metalloid{"Silicon", 14, 28.085},
    metalloid{"Germanium", 32, 74.63},
    metalloid{"Arsenic", 33, 74.921},
    metalloid{"Antimony", 51, 121.760},
    metalloid{"Tellerium", 52, 127.60},
    metalloid{"Polonium", 84, 209.0},
}
```

Go also supports a `struct` type which is a composite that stores named elements called fields as shown in the following code:

```
func main() {
    planet := struct {
        name string
        diameter int
    }{"earth", 12742}
}
```

The previous example uses the literal syntax to declare `struct{name string; diameter int}` with the value `{"earth", 12742}`. You can read all about composite types in Chapter 7, *Composite Types*.

The named type

As discussed, Go provides a healthy set of built-in types, both simple and composite. Go programmers can also define new named types based on an existing underlying type as shown in the following snippet extracted from `metalloid` in the earlier example:

```
type amu float64

type metalloid struct {
    name string
    number int32
    weight amu
}
```

The previous snippet shows the definition of two named types, one called `amu`, which uses type `float64` as its underlying type. Type `metalloid`, on the other hand, uses a `struct` composite type as its underlying type, allowing it to store values in an indexed data structure. You can read more about declaring new named types in [Chapter 4, Data Types](#).

Methods and objects

Go is not an object-oriented language in a classical sense. Go types do not use a class hierarchy to model the world as is the case with other object-oriented languages. However, Go can support the object-based development idiom, allowing data to receive behaviors. This is done by attaching functions, known as methods, to named types.

The following snippet, extracted from the `metalloid` example, shows the type `amu` receiving a method called `float()` that returns the mass as a `float64` value:

```
type amu float64

func (mass amu) float() float64 {
    return float64(mass)
}
```

The power of this concept is explored in detail in [Chapter 8, Methods, Interfaces, and Objects](#).

Interfaces

Go supports the notion of a programmatic interface. However, as you will see in Chapter 8, *Methods, Interfaces, and Objects*, the Go interface is itself a type that aggregates a set of methods that can project capabilities onto values of other types. Staying true to its simplistic nature, implementing a Go interface does not require a keyword to explicitly declare an interface. Instead, the type system implicitly resolves implemented interfaces using the methods attached to a type.

For instance, Go includes the built-in interface called `Stringer`, defined as follows:

```
type Stringer interface {
    String() string
}
```

Any type that has the method `String()` attached, automatically implements the `Stringer` interface. So, modifying the definition of the type `metalloid`, from the previous program, to attach the method `String()` will automatically implement the `Stringer` interface:

```
type metalloid struct {
    name string
    number int32
    weight amu
}
func (m metalloid) String() string {
    return fmt.Sprintf(
        "%-10s %-10d %-10.3f %e",
        m.name, m.number, m.weight.float(), atoms(moles(m.weight)),
    )
}
```

golang.fyi/ch01/metalloids2.go

The `String()` methods return a pre-formatted string that represents the value of a `metalloid`. The function `Print()`, from the standard library package `fmt`, will automatically call the method `String()`, if its parameter implements `stringer`. So, we can use this fact to print `metalloid` values as follow:

```
func main() {
    fmt.Print(headers())
    for _, m := range metalloids {
        fmt.Print(m, "\n")
    }
}
```

Again, refer to Chapter 8, *Methods, Interfaces, and Objects*, for a thorough treatment of the topic of interfaces.

Concurrency and channels

One of the main features that has rocketed Go to its current level of adoption is its inherent support for simple concurrency idioms. The language uses a unit of concurrency known as a `goroutine`, which lets programmers structure programs with independent and highly concurrent code.

As you will see in the following example, Go also relies on a construct known as a channel used for both communication and coordination among independently running `goroutines`. This approach avoids the perilous and (sometimes brittle) traditional approach of thread communicating by sharing memory. Instead, Go facilitates the approach of sharing by communicating using channels. This is illustrated in the following example that uses both `goroutines` and channels as processing and communication primitives:

```
// Calculates sum of all multiple of 3 and 5 less than MAX value.
// See https://projecteuler.net/problem=1
package main

import (
    "fmt"
)

const MAX = 1000

func main() {
    work := make(chan int, MAX)
    result := make(chan int)

    // 1. Create channel of multiples of 3 and 5
    // concurrently using goroutine
    go func(){
        for i := 1; i < MAX; i++ {
            if (i % 3) == 0 || (i % 5) == 0 {
                work <- i // push for work
            }
        }
        close(work)
    }()
}

// 2. Concurrently sum up work and put result
//     in channel result
go func(){
    var sum int
    for i := range work {
        sum += i
    }
    result <- sum
}
```

```
r := 0
for i := range work {
    r = r + i
}
result <- r
}()

// 3. Wait for result, then print
fmt.Println("Total:", <- result)
}
```

golang.fyi/ch01/euler1.go

The code in the previous example splits the work to be done between two concurrently running goroutines (declared with the `go` keyword) as annotated in the code comment. Each goroutine runs independently and uses the Go channels, `work` and `result`, to communicate and coordinate the calculation of the final result. Again, if this code does not make sense at all, rest assured, concurrency has the whole of Chapter 9, *Concurrency*, dedicated to it.

Memory management and safety

Similar to other compiled and statically-typed languages such as C and C++, Go lets developers have direct influence on memory allocation and layout. When a developer creates a `slice` (think `array`) of bytes, for instance, there is a direct representation of those bytes in the underlying physical memory of the machine. Furthermore, Go borrows the notion of pointers to represent the memory addresses of stored values giving Go programs the support of passing function parameters by both value and reference.

Go asserts a highly opinionated safety barrier around memory management with little to no configurable parameters. Go automatically handles the drudgery of bookkeeping for memory allocation and release using a runtime garbage collector. Pointer arithmetic is not permitted at runtime; therefore, developers cannot traverse memory blocks by adding to or subtracting from a base memory address.

Fast compilation

Another one of Go's attractions is its millisecond build-time for moderately-sized projects. This is made possible with features such as a simple syntax, conflict-free grammar, and a strict identifier resolution that forbids unused declared resources such as imported packages or variables. Furthermore, the build system resolves packages using transitivity information stored in the closest source node in the dependency tree. Again, this reduces the code-compile-run cycle to feel more like a dynamic language instead of a compiled language.

Testing and code coverage

While other languages usually rely on third-party tools for testing, Go includes both a built-in API and tools designed specifically for automated testing, benchmarking, and code coverage. Similar to other features in Go, the test tools use simple conventions to automatically inspect and instrument the test functions found in your code.

The following function is a simplistic implementation of the Euclidean division algorithm that returns a quotient and a remainder value (as variables `q` and `r`) for positive integers:

```
func DivMod(dvnd, dvsr int) (q, r int) {
    r = dvnd
    for r >= dvsr {
        q += 1
        r = r - dvsr
    }
    return
}
```

golang.fyi/ch01/testexample/divide.go

In a separate source file, we can write a test function to validate the algorithm by checking the remainder value returned by the tested function using the Go test API as shown in the following code:

```
package testexample
import "testing"
func TestDivide(t *testing.T) {
    dvnd := 40
    for dvsor := 1; dvsor < dvnd; dvsor++ {
        q, r := DivMod(dvnd, dvsor)
        if (dvnd % dvsor) != r {
            t.Fatalf("%d/%d q=%d, r=%d, bad remainder.", dvnd, dvsor, q, r)
        }
}
```

```
}
```

golang.fyi/ch01/testexample/divide_test.go

To exercise the test source code, simply run Go's test tool as shown in the following example:

```
$> go test .
ok    github.com/vladimirvivien/learning-go/ch01/testexample 0.003s
```

The test tool reports a summary of the test result indicating the package that was tested and its pass/fail outcome. The Go Toolchain comes with many more features designed to help programmers create testable code, including:

- Automatically instrument code to gather coverage statistics during tests
- Generating HTML reports for covered code and tested paths
- A benchmark API that lets developers collect performance metrics from tests
- Benchmark reports with valuable metrics for detecting performance issues

You can read all about testing and its related tools in Chapter 12, *Code Testing*.

Documentation

Documentation is a first-class component in Go. Arguably, the language's popularity is in part due to its extensive documentation (see <http://golang.org/pkg>). Go comes with the Godoc tool, which makes it easy to extract documentation from comment text embedded directly in the source code. For example, to document the function from the previous section, we simply add comment lines directly above the `DivMod` function as shown in the following example:

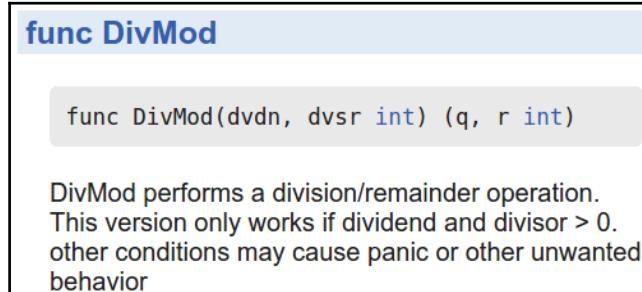
```
// DivMod performs a Euclidian division producing a quotient and remainder.
// This version only works if dividend and divisor > 0.
func DivMod(dvdn, dvsr int) (q, r int) {
    ...
}
```

The Go documentation tool can automatically extract and create HTML-formatted pages. For instance, the following command will start the Godoc tool as a server on `localhost` port 6000:

```
$> godoc -http=:6001
```

You can then access the documentation of your code directly from your web browser. For instance, the following figure shows the generated documentation snippet for the previous function located at

<http://localhost:6001/pkg/github.com/vladimirvivien/learning-go/ch01/testexample/>:



An extensive library

For its short existence, Go rapidly grew a collection of high-quality APIs as part of its standard library that are comparable to other popular and more established languages. The following, by no means exhaustive, lists some of the core APIs that programmers get out-of-the-box:

- Complete support for regular expressions with search and replace
- Powerful IO primitives for reading and writing bytes
- Full support for networking from socket, TCP/UDP, IPv4, and IPv6
- APIs for writing production-ready HTTP services and clients
- Support for traditional synchronization primitives (mutex, atomic, and so on)
- General-purpose template framework with HTML support
- Support for JSON/XML serializations
- RPC with multiple wire formats
- APIs for archive and compression algorithms: `tar`, `zip/gzip`, `zlib`, and so on
- Cryptography support for most major algorithms and hash functions
- Access to OS-level processes, environment info, signaling, and much more

The Go Toolchain

Before we end the chapter, one last aspect of Go that should be highlighted is its collection of tools. While some of these tools were already mentioned in previous sections, others are listed here for your awareness:

- `fmt`: Reformats source code to adhere to the standard
- `vet`: Reports improper usage of source code constructs
- `lint`: Another source code tool that reports flagrant style infractions
- `goimports`: Analyzes and fixes package import references in source code
- `godoc`: Generates and organizes source code documentation
- `generate`: Generates Go source code from directives stored in source code
- `get`: Remotely retrieves and installs packages and their dependencies
- `build`: Compiles code in a specified package and its dependencies
- `run`: Provides the convenience of compiling and running your Go program
- `test`: Performs unit tests with support for benchmark and coverage reports
- `oracle` static analysis tool: Queries source code structures and elements
- `cgo`: Generates source code for interoperability between Go and C

Summary

Within its relatively short existence, Go has won the hearts of many adopters who value simplicity as a way to write code that is exact and is able to scale in longevity. As you have seen from the previous sections in this chapter, it is easy to get started with your first Go program.

The chapter also exposed its readers to a high-level summary of the most essential features of Go including its simplified syntax, its emphasis on concurrency, and the tools that make Go a top choice for software engineers, creating systems for the age of data center computing. As you may imagine, this is just a taste of what's to come.

In the following chapters, the book will continue to explore in detail the syntactical elements and language concepts that make Go a great language to learn. Let's Go!

2

Go Language Essentials

In the previous chapter, we established the elemental characteristics that make Go a great language with which to create modern system programs. In this chapter, we dig deeper into the language's syntax to explore its components and features.

We will cover the following topics:

- The Go source file
- Identifiers
- Variables
- Constants
- Operators

The Go source file

We have seen, in Chapter 1, *A First Step in Go*, some examples of Go programs. In this section, we will examine the Go source file. Let us consider the following source code file (which prints "Hello World" greetings in different languages):

```
package main

import "fmt"
import "math/rand"
import "time"

var greetings = [][]string{
    {"Hello, World!", "English"},
    {"Salut Monde", "French"},
    {"世界您好", "Simplified Chinese"},
    {"qo' vIvan", "Klingon"},
    {"हैलो वर्ल्ड", "Hindi"},
    {"안녕하세요", "Korean"},
    {"привет мир", "Russian"},
    {"Wapendwa Dunia", "Swahili"},
    {"Hola Mundo", "Spanish"},
    {"Merhaba Dünya", "Turkish"},
}

func greeting() [] string {
    seed := time.Now().UnixNano()
    rnd := rand.New(rand.NewSource(seed))
    return greetings[rnd.Intn(len(greetings))]
}

func main() {
    g := greeting()
    fmt.Printf("%s (%s)\n", g[0], g[1])
}
```

golang.fyi/ch02/helloworld2.go

A typical Go source file, such as the one listed earlier, can be divided into three main sections, illustrated as follows:

- The Package Clause:

```
//1 Package Clause
package main
```

- **The Import Declaration:**

```
//2 Import Declaration
import "fmt"
import "math/rand"
import "time"
```

- **The Source Body:**

```
//3 Source Body
var greetings = [][]string{
    {"Hello, World!", "English"},
    ...
}

func greeting() [] string {
    ...
}

func main() {
    ...
}
```

The **package** clause indicates the name of the package this source file belongs to (see Chapter 6, *Go Packages and Programs* for a detailed discussion on package organization). The **import** declaration lists any external package that the source code wishes to use. The Go compiler strictly enforces package declaration usage. It is considered an error (compilation) to include an unused package in your source file. The last portion of the source is considered the body of your source file. It is where you declare variables, constants, types, and functions.

All Go source files must end with the `.go` suffix. In general, you can name a Go source file whatever you want. Unlike Java, for instance, there is no direct association between a Go file name and the types it declared in its content. It is, however, considered good practice to name your file something indicative of its content.

Before we explore Go's syntax in greater detail, it is important to understand some basic structural elements of the language. While some of these elements are syntactically bolted into the language, others are simple idioms and conventions that you should be aware of to make your introduction to Go simple and enjoyable.

Optional semicolon

You may have noticed that Go does not require a semicolon as a statement separator. This is a trait borrowed from other lighter and interpreted languages. The following two programs are functionally equivalent. The first program uses idiomatic Go and omits the semicolons:

```
package main
import "fmt"
func main() {
    fmt.Println("Hello, 世界")
}
```

The second version of the program, shown as follows, uses superfluous semicolons to explicitly terminate its statements. While the compiler may thank you for your help, this is not idiomatic in Go:

```
package main;
import "fmt";

func main() {
    fmt.Println("Hello, 世界");
}
```

Although semicolons in Go are optional, Go's formal grammar still requires them as statement terminators. So, the Go compiler will insert semicolons at the end of source code lines that end with the following:

- An identifier
- A literal value for string, Boolean, numeric, or complex
- A control flow directive such as break, continue, or return
- A closing parenthesis or bracket such as), }, or]
- The increment ++ or the decrement -- operator

Due to these rules, the compiler enforces strict syntactical forms that heavily influence source code style in Go. For instance, all code blocks must start with an open curly { brace on the same line as its preceding statement. Otherwise, the compiler may insert the semicolon in a location that breaks the code, as shown in the following `if` statement:

```
func main() {
    if "a" == "a"
    {
        fmt.Println("Hello, World!")
    }
}
```

Moving the curly brace to the next line causes the compiler to insert the semicolon prematurely, which will result in the following syntax error:

```
$> ... missing condition in if statement ...
```

This is because the compiler inserted the semicolon after the `if` statement (`if "a"=="a";`), using the semicolon insertion rules discussed in this section. You can verify this by manually inserting a semicolon after the `if` condition statement; you will get the same error. This is an excellent place to transition into the next section, to discuss trailing commas in code blocks.

Multiple lines

Breaking up expressions into multiple lines must follow the semi-colon rules discussed in the previous section. Mainly, in a multi-line expression, each line must end with a token that prevents the premature insertion of a semi-colon, as illustrated in the following table. It should be noted that rows in the table with an invalid expression will not compile:

Expression	Valid
<code>lonStr := "Hello World! " + "How are you?"</code>	Yes, the + operator prevents a premature semi-colon from being inserted.
<code>lonStr := "Hello World! " + "How are you?"</code>	No, a semi-colon will be inserted after the first line, semantically breaking the line.
<code>fmt.Printf("[%s] %d %d %v", str, num1, num2, nameMap)</code>	Yes, the comma properly breaks the expression.

<pre>fmt.Printf("[%s] %d %d %v", str, num1, num2, nameMap)</pre>	Yes, the compiler inserts a semi-colon only after the last line.
<pre>weekDays := []string{ "Mon", "Tue", "Wed", "Thr", "Fri" }</pre>	No, the Fri line causes a premature semi-colon to be inserted.
<pre>weekDays2 := []string{ "Mon", "Tue", "Wed", "Thr", "Fri", }</pre>	Yes, the Fri line contains a trailing comma, which causes compiler to insert a semi-colon at the next line.
<pre>weekDays1 := []string{ "Mon", "Tue", "Wed", "Thr", "Fri"}</pre>	Yes, the semi-colon is inserted after the line with the closing bracket.

You may wonder why the Go compiler puts the onus on the developer to provide line-break hints to indicate the end of a statement. Surely, Go designers could have devised an elaborate algorithm to figure this out automatically. Yes, they could have. However, by keeping the syntax simple and predictable, the compiler is able to quickly parse and compile Go source code.



The Go toolchain includes the `gofmt` tool, which can be used to consistently apply proper formatting rules to your source code. There is also the `govet` tool, which goes much further by analyzing your code for structural problems with code elements.

Go identifiers

Go identifiers are used to name program elements including packages, variables, functions, and types. The following summarizes some attributes about identifiers in Go:

- Identifiers support the Unicode character set
- The first position of an identifier must be a letter or an underscore
- Idiomatic Go favors mixed caps (camel case) naming
- Package-level identifiers must be unique across a given package
- Identifiers must be unique within a code block (functions, control statements)

The blank identifier

The Go compiler is particularly strict about the use of declared identifiers for variables or packages. The basic rule is: *you declare it, you must use it*. If you attempt to compile code with unused identifiers such as variables or named packages, the compilers will not be pleased and will fail compilation.

Go allows you to turn off this behavior using the blank identifier, represented by the `_` (underscore) character. Any declaration or assignment that uses the blank identifier is not bound to any value and is ignored at compile time. The blank identifier is usually used in two contexts, as listed in the following subsections.

Muting package imports

When a package declaration is preceded by an underscore, the compiler allows the package to be declared without any further referenced usage:

```
import "fmt"
import "path/filepath"
import _ "log"
```

In the previous code snippet, the package `log` will be muted without any further reference in the code. This can be a handy feature during active development of new code, where developers may want to try new ideas without constantly having to comment out or delete the declarations. Although a package with a blank identifier is not bound to any reference, the Go runtime will still initialize it. Chapter 6, *Go Packages and Programs*, discusses the package initialization lifecycle.

Muting unwanted function results

When a Go function call returns multiple values, each value in the return list must be assigned to a variable identifier. In some cases, however, it may be desirable to mute unwanted results from the return list while keeping others, as shown in the following call:

```
_ , execFile := filepath.Split("/opt/data/bigdata.txt")
```

The previous call to the function `filepath.Split("/opt/data/bigdata.txt")` takes a path and returns two values: the first is the parent path (`/opt/data`) and the second is the file name (`bigdata.txt`). The first value is assigned to the blank identifier and is, therefore, unbounded to a named identifier, which causes it to be ignored by the compiler. In future discussions, we will explore other uses of this idiom's other contexts, such as error-handling and `for` loops.

Built-in identifiers

Go comes with a number of built-in identifiers. They fall into different categories, including types, values, and built-in function.

Types

The following identifiers are used for Go's built-in types:

Category	Identifier
Numeric	<code>byte, int, int8, int16, int32, int64, rune, uint, uint8, uint16, uint32, uint64, float32, float64, complex64, complex128, uintptr</code>
String	<code>string</code>
Boolean	<code>bool</code>
Error	<code>error</code>

Values

These identifiers have preassigned values:

Category	Identifier
Boolean constants	true, false
Constant counter	iota
Uninitialized value	nil

Functions

The following functions are available as part of Go's built-in pre-declared identifiers:

Category	Identifier
Initialization	make(), new()
Collections	append(), cap(), copy(), delete()
Complex numbers	complex(), imag(), real()
Error Handling	panic(), recover()

Go variables

Go is a strictly typed language, which implies that all variables are named elements that are bound to both a value and a type. As you will see, the simplicity and flexibility of its syntax make declaring and initializing variables in Go feel more like a dynamically-typed language.

Variable declaration

Before you can use a variable in Go, it must be declared with a named identifier for future reference in the code. The long form of a variable declaration in Go follows the format shown here:

```
var <identifier list> <type>
```

The `var` keyword is used to declare one or more variable identifiers followed by the type of the variables. The following source code snippet shows an abbreviated program with several variables declared outside of the function `main()`:

```
package main

import "fmt"

var name, desc string
var radius int32
var mass float64
var active bool
var satellites []string

func main() {
    name = "Sun"
    desc = "Star"
    radius = 685800
    mass = 1.989E+30
    active = true
    satellites = []string{
        "Mercury",
        "Venus",
        "Earth",
        "Mars",
        "Jupiter",
        "Saturn",
        "Uranus",
        "Neptune",
    }
    fmt.Println(name)
    fmt.Println(desc)
    fmt.Println("Radius (km)", radius)
    fmt.Println("Mass (kg)", mass)
    fmt.Println("Satellites", satellites)
}
```

golang.fyi/ch02/vardec1.go

The zero-value

The previous source code shows several examples of variables being declared with a variety of types. Then the variables are assigned a value inside the function `main()`. At first glance, it would appear that these declared variables do not have an assigned value when they are declared. This would contradict our previous assertion that all Go variables are bound to a type and a value.

How can we declare a variable and not bind a value to it? During declaration of a variable, if a value is not provided, Go will automatically bind a default value (or a zero-value) to the variable for proper memory initialization (we see how to do both declaration and initialization in one expression later).

The following table shows Go types and their default zero-values:

Type	Zero-Value
<code>string</code>	<code>" "</code> (empty string)
Numeric - Integers: <code>byte</code> , <code>int</code> , <code>int8</code> , <code>int16</code> , <code>int32</code> , <code>int64</code> , <code>rune</code> , <code>uint</code> , <code>uint8</code> , <code>uint16</code> , <code>uint32</code> , <code>uint64</code> , <code>uintptr</code>	<code>0</code>
Numeric - Floating point: <code>float32</code> , <code>float64</code>	<code>0.0</code>
<code>bool</code>	<code>false</code>
Array	Each index position has a zero-value corresponding to the array's element type.
Struct	An empty struct with each member having its respective zero-value.
Other types: Interface, function, channel, slice, map, and pointer	<code>nil</code>

Initialized declaration

As hinted earlier, Go also supports the combination of both variable declaration and initialization as one expression using the following format:

var <identifier list> <type> = <value list or initializer expressions>

This declaration format has the following properties:

- An identifier list provided on the left-hand side of the equal sign (followed by a type)
- A matching comma-separated value list on the right-hand side
- Assignment occurs in the respective order of identifiers and values
- Initializer expressions must yield a matching list of values

The following abbreviated example shows the declaration and initialization combination at work:

```
var name, desc string = "Earth", "Planet"
var radius int32 = 6378
var mass float64 = 5.972E+24
var active bool = true
var satellites = []string{
    "Moon",
}
```

golang.fyi/ch02/vardec2.go

Omitting variable types

So far, we have discussed what is called the long form of Go's variable declaration and initialization. To make the language feel closer to its dynamically-typed cousins, the type specification can be omitted, as shown in the following declaration format:

var <identifier list> = <value list or initializer expressions>

During compilation, the compiler infers the type of the variable based on the assigned value or the initializer expression on the right-hand side of the equal sign, as shown in the following example.

```
var name, desc = "Mars", "Planet"
var radius = 6755
var mass = 6416930000000000.0
var active = true
var satellites = []string{
    "Phobos",
    "Deimos",
}
```

golang.fyi/ch02/vardec3.go

As stated earlier, when a variable is assigned a value, it must receive a type along with that value. When the type of the variable is omitted, the type information is deduced from the assigned value or the returned value of an expression. The following table shows the type that is inferred given a literal value:

Literal value	Inferred type
Double- or single-quoted (raw) text: "Planet Mars" "All planets revolve around the Sun."	string
Integers: -76 0 1244 1840	int
Decimals: -0.25 4.0 3.1e4 7e-12	float64
Complex numbers: -5.0i 3i (0+4i)	complex128
Booleans: true false	bool
Array values: [2]int{-76, 8080}	The array type defined in the literal value. In this case it is: [2]int
Map values: map[string]int{ "Sun": 685800, "Earth": 6378, "Mars": 3396, }	The map type defined in the literal value. In this case it is: map[string]int
Slice values: []int{-76, 0, 1244, 1840}	The slice type defined in the literal value: []int

Struct values: <pre>struct{ name string diameter int} { "Mars", 3396, }</pre>	A struct type as defined in the literal value. In this case the type is: <pre>struct{name string; diameter int}</pre>
Function values: <pre>var sqr = func (v int) int { return v * v }</pre>	The function type defined in the function definition literal. In this case, variable <code>sqr</code> will have type: <pre>func (v int) int</pre>

Short variable declaration

Go can further reduce the variable declaration syntax using the *short variable declaration* format. In this format, the declaration loses the var keyword and the type specification, and uses an assignment operator := (colon-equal), as shown:

<identifier list> := <value list or initializer expressions>

This is a simple and uncluttered idiom that is commonly used when declaring variables in Go. The following code sample shows usage of the short variable declarations:

```
func main() {
    name := "Neptune"
    desc := "Planet"
    radius := 24764
    mass := 1.024e26
    active := true
    satellites := []string{
        "Naiad", "Thalassa", "Despina", "Galatea", "Larissa",
        "S/2004 N 1", "Proteus", "Triton", "Nereid", "Halimede",
        "Sao", "Laomedea", "Neso", "Psamathe",
    }
    ...
}
```

golang.fyi/ch02/vardec4.go

Notice the keyword `var` and variable types have been omitted in the declaration. Short variable declaration uses the same mechanism to infer the type of the variable discussed earlier.

Restrictions for short variable declaration

For convenience, the short form of the variable declaration does come with several restrictions that you should be aware of to avoid confusion:

- Firstly, it can only be used within a function block
- The assignment operator `:=`, declares variable and assign values
- `:=` cannot be used to update a previously declared variable
- Updates to variables must be done with an equal sign

While these restrictions may have their justifications rooted in the simplicity of Go's grammar, they are generally viewed as a source of confusion for newcomers to the language. For instance, the colon-equal operator cannot be used with package-level variables assignments. Developers learning Go may find it compelling to use the assignment operator as a way to update a variable, but that would cause a compilation error.

Variable scope and visibility

Go uses lexical scoping based on code blocks to determine the visibility of variables within a package. Depending on the location where a variable is declared, within the source text, will determine its scope. As a general rule, a variable is only accessible from within the block where it is declared and visible to all nested sub-blocks.

The following screenshot illustrates the scope of several variables declared within a source text. Each variable declaration is marked with its scope (package, function, for loop, and if...else block):

```
import (
    "bytes"
    ...
    "strconv"
)

var mapFile string = "./nummap.txt"
var numbersFile = "./nums.txt"
var fileMode = 4000
var nums bytes.Buffer

func loadNumberMap() error {
    data, err := ioutil.ReadFile(mapFile)
    if err != nil {
        return err
    }
    for i, b := range data {
        if rune(b) == '1' {
            nums.WriteString(strconv.Itoa(i))
            nums.WriteRune('\n')
        }
    }
    fmt.Println("Loaded all mapped values.")
    return nil
}
```

The code snippet shows variable declarations and their scopes. Variables `mapFile`, `numbersFile`, and `fileMode` are at the package level. Variable `nums` is declared within the `loadNumberMap` function. Inside the function, `data` and `err` are function-level variables. The `for` loop introduces `i` and `b`, which are local to the loop block.

golang.fyi/ch02/makenums.go

As explained earlier, variable visibility works top-down. Variables with package scope, such as `mapFile` and `numbersFile`, are globally visible to all other elements in the package. Moving down the scope ladder, function-block variables such as `data` and `err` are visible to all elements in the function and including sub-blocks. Variables `i` and `b` in the inner `for` loop block are only visible within that block. Once the loop is done, `i` and `b` would go out of scope.

One source of confusion to newcomers to Go is the visibility of package-scoped variables. When a variable is declared at package level (outside of a function or method block), it is globally visible to the entire package, not just to the source file where the variable is declared. This means a package-scoped variable identifier can only be declared once in a group of files that make up a package, a fact that may not be obvious to developers starting out with Go. Refer to Chapter 6, *Go Packages and Programs*, for details on package organization.



Variable declaration block

Go's syntax allows the declaration of top-level variables to be grouped together into blocks for greater readability and code organization. The following example shows a rewrite of one of the previous examples using the variable declaration block:

```
var (
    name string = "Earth"
    desc string = "Planet"
    radius int32 = 6378
    mass float64 = 5.972E+24
    active bool = true
    satellites []string
)
```

golang.fyi/ch02/vardec5.go

Go constants

In Go, a constant is a value with a literal representation such as a string of text, Boolean, or numbers. The value for a constant is static and cannot be changed after initial assignment. While the concept they represent is simple, constants, however, have some interesting properties that make them useful, especially when working with numeric values.

Constant literals

Constants are values that can be represented by a text literal in the language. One of the most interesting properties of constants is that their literal representations can either be treated as typed or untyped values. Unlike variables, which are intrinsically bound to a type, constants can be stored as untyped values in memory space. Without that type constraint, numeric constant values, for instance, can be stored with great precision.

The followings are examples of valid constant literal values that can be expressed in Go:

```
"Mastering Go"
'G'
false
111009
2.71828
94314483457513374347558557572455574926671352 1e+500
5.0i
```

Typed constants

Go constant values can be bound to named identifiers using a constant declaration. Similar to a variable declaration, Go uses the `const` keyword to indicate the declaration of a constant. Unlike variables, however, the declaration must include the literal value to be bound to the identifier, as shown in the following format:

const <identifier list> type = <value list or initializer expressions>

Constants cannot have any dependency that requires runtime resolution. The compiler must be able to resolve the value of a constant at compile time. This means all constants must be declared and initialized with a value literal (or an expression that results to a constant value).

The following snippet shows some typed constants being declared:

```
const a1, a2 string = "Mastering", "Go"
const b rune = 'G'
const c bool = false
const d int32 = 111009
const e float32 = 2.71828
const f float64 = math.Pi * 2.0e+3
const g complex64 = 5.0i
const h time.Duration = 4 * time.Second
```

golang.fyi/ch02/const.go

Notice in the previous source snippet that each declared constant identifier is explicitly given a type. As you would expect, this implies that the constant identifier can only be used in contexts that are compatible with its types. However, the next section explains how this works differently when the type is omitted in the constant declaration.

Untyped constants

Constants are even more interesting when they are untyped. An untyped constant is declared as follows:

const <identifier list> = <value list or initializer expression>

As before, the keyword `const` is used to declare a list of identifiers as constants along with their respective bounded values. However, in this format, the type specification is omitted in the declaration. As an untyped entity, a constant is merely a blob of bytes in memory without any type precision restrictions imposed. The following shows some sample declarations of untyped constants:

```
const i = "G is" + " for Go "
const j = 'V'
const k1, k2 = true, !k1
const l = 111*100000 + 9
const m1 = math.Pi / 3.141592
const m2 = 1.414213562373095048801688724209698078569671875376...
const m3 = m2 * m2
const m4 = m3 * 1.0e+400
const n = -5.0i * 3
const o = time.Millisecond * 5
```

golang.fyi/ch02/const.go

From the previous code snippet, the untyped constant `m2` is assigned a long decimal value (truncated to fit on the printed page as it goes another 17 digits). Constant `m4` is assigned a much larger number of $m3 \times 1.0e+400$. The entire value of the resulting constant is stored in memory without any loss in precision. This can be an extremely useful tool for developers interested in computations where a high level of precision is important.

Assigning untyped constants

Untyped constant values are of limited use until they are assigned to variables, used as function parameters, or are part of an expression assigned to a variable. In a strongly-typed language like Go, this means there is a potential for some type adjustment to ensure that the value stored in the constant can be properly assigned to the target variable. One advantage of using untyped constants is that the type system relaxes the strict application of type checking. An untyped constant can be assigned to different, though compatible, types of different precision without any complaint from the compiler, as shown in the following example:

```
const m2 = 1.414213562373095048801688724209698078569671875376...
var u1 float32 = m2
var u2 float64 = m2
u3 := m2
```

The previous snippet shows the untyped constant `m2` being assigned to two variables of different floating-point precisions, `u1` and `u2`, and to an untyped variable, `u3`. This is possible because constant `m2` is stored as a raw untyped value and can therefore be assigned to any variable compatible with its representation (a floating point).

While the type system will accommodate the assignment of `m2` to variables of different precision, the resulting assignment is adjusted to fit the variable type, as noted in the following:

```
u1 = 1.4142135      //float32
u2 = 1.4142135623730951 //float64
```

What about variable `u3`, which is itself an untyped variable? Since `u3` does not have a specified type, it will rely on type inference from the constant value to receive a type assignment. Recall from the discussion in the section *Omitting Variable Types* earlier, that constant literals are mapped to basic Go types based on their textual representations. Since constant `m2` represents a decimal value, the compiler will infer its default to be a `float64`, which will be automatically assigned to variable `u3`, as shown:

```
u3 = 1.4142135623730951 //float64
```

As you can see, Go's treatment of untyped raw constant literals increases the language's usability by automatically applying some simple, but effective, type inference rules without sacrificing type-safety. Unlike other languages, developers do not have to explicitly specify the type in the value literal or perform some sort of typecast to make this work.

Constant declaration block

As you may have guessed, constant declarations, can be organized as code blocks to increase readability. The previous example can be rewritten as follows:

```
const (
    a1, a2 string      = "Mastering", "Go"
    b       rune        = 'G'
    c       bool         = false
    d       int32        = 111009
    e       float32      = 2.71828
    f       float64      = math.Pi * 2.0e+3
    g       complex64     = 5.0i
    h       time.Duration = 4 * time.Second
    ...
)
```

golang.fyi/ch02/const2.go

Constant enumeration

One interesting usage of constants is to create enumerated values. Using the declaration block format (shown in the preceding section), you can easily create numerically increasing enumerated integer values. Simply assign the pre-declared constant value `iota` to a constant identifier in the declaration block, as shown in the following code sample:

```
const (
    StarHyperGiant = iota
    StarSuperGiant
    StarBrightGiant
    StarGiant
    StarSubGiant
    StarDwarf
    StarSubDwarf
    StarWhiteDwarf
    StarRedDwarf
    StarBrownDwarf
)
```

golang.fyi/ch02/enum0.go

The compiler will then automatically do the following:

- Declare each member in the block as an untyped integer constant value
- Initialize `iota` with a value of zero
- Assign `iota`, or zero, to the first constant member (`StarHyperGiant`)
- Each subsequent constant is assigned an `int` value increased by one

So the previous list of constants would be assigned a sequence of values going from zero to nine. Whenever `const` appears as a declaration block, it resets the counter to zero. In the following snippet, each set of constants is enumerated from zero to four separately:

```
const (
    StarHyperGiant = iota
    StarSuperGiant
    StarBrightGiant
    StarGiant
    StarSubGiant
)
const (
    StarDwarf = iota
    StarSubDwarf
    StarWhiteDwarf
    StarRedDwarf
)
```

```
StarBrownDwarf  
)  
  
golang.fyi/ch02/enum1.go
```

Overriding the default enumeration type

By default, an enumerated constant is declared as an untyped integer value. However, you can override the default type of the enumerated values by providing an explicit numeric type for your enumerated constants, as shown in the following code sample:

```
const (  
    StarDwarf byte = iota  
    StarSubDwarf  
    StarWhiteDwarf  
    StarRedDwarf  
    StarBrownDwarf  
)
```

You can specify any numeric type that can represent integers or floating point values. For instance, in the preceding code sample, each constant will be declared as type `byte`.

Using iota in expressions

When `iota` appears in an expression, the same mechanism works as expected. The compiler will apply the expression for each successive increasing value of `iota`. The following example assigns even numbers to the enumerated members of the constant declaration block:

```
const (  
    StarHyperGiant = 2.0*iota  
    StarSuperGiant  
    StarBrightGiant  
    StarGiant  
    StarSubGiant  
)  
  
golang.fyi/ch02/enum2.go
```

As you may expect, the previous example assigns an even value to each enumerated constants, starting with 0, as shown in the following output:

```
StarHyperGiant = 0      [float64]
```

```
StarSuperGiant = 2      [float64]
StarBrightGiant = 4     [float64]
StarGiant = 6           [float64]
StarSubGiant = 8        [float64]
```

Skipping enumerated values

When working with enumerated constants, you may want to throw away certain values that should not be part of the enumeration. This can be accomplished by assigning iota to the blank identifier at the desired position in the enumeration. For instance, the following skips the values 0 and 64:

```
_ = iota    // value 0
StarHyperGiant = 1 << iota
StarSuperGiant
StarBrightGiant
StarGiant
StarSubGiant
_           // value 64
StarDwarf
StarSubDwarf
StarWhiteDwarf
StarRedDwarf
StarBrownDwarf
```

golang.fyi/ch02/enum3.go

Since we skip iota position 0, the first assigned constant value is at position 1. This results in expression `1 << iota` resolving to `1 << 1 = 2`. The same is done at the sixth position, where expression `1 << iota` returns 64. That value will be skipped and not assigned to any constant, as shown in the following output:

```
StarHyperGiant = 2
StarSuperGiant = 4
StarBrightGiant = 8
StarGiant = 16
StarSubGiant = 32
StarDwarf = 128
StarSubDwarf = 256
StarWhiteDwarf = 512
StarRedDwarf = 1024
StarBrownDwarf = 2048
```

Go operators

Staying true to its simplistic nature, operators in Go do exactly what you would expect, mainly, they allow operands to be combined into expressions. There are no hidden surprise behaviors with Go operators as there is no support for operator-overloading as found in C++ or Scala. This was a deliberate decision from the designers to keep the semantics of the language simple and predictable.

This section explores the most common operators that you will encounter as you start with Go. Other operators are covered throughout other chapters of the book.

Arithmetic operators

The following table summarizes the arithmetic operators supported in Go.

Operator	Operation	Compatible types
<code>*</code> , <code>/</code> , <code>-</code>	Multiplication, division, and subtraction	Integers, floating points, and complex numbers
<code>%</code>	Remainder	Integers
<code>+</code>	Addition	Integers, floating points, complex numbers, and strings (concatenation)

Note that the addition operator, `+`, can be applied to strings such as in the expression `var i = "G is" + " for Go"`. The two string operands are concatenated to create a new string that is assigned to variable `i`.

The increment and decrement operators

As with other C-like languages, Go supports the `++` (increment) and the `--` (decrement) operators. When applied, these operators increase, or decrease, the operand's value by one, respectively. The following shows a function that uses the decrement operator to traverse the letters in string `s` in the reverse order:

```
func reverse(s string) {
    for i := len(s) - 1; i >= 0; {
        fmt.Println(string(s[i]))
        i--
    }
}
```

It is important to note that the increment and decrement operators are statements, not expressions, as shown in the following snippets:

```
nextChar := i++      // syntax error
fmt.Println("Current char", i--)    // syntax error
nextChar++          // OK
```

In the preceding examples, it is worth noting that the increment and decrement statements only support the postfix notation. The following snippet would not compile because of statement `-i`:

```
for i := len(s) - 1; i >= 0; {
    fmt.Println(string(s[i]))
    --i    //syntax error
}
```

Go assignment operators

Operator	Description
=	The simple assignment works as expected. It updates the left operand with the value of the right.
:=	The colon-equal operator declares a new variable, the left-side operator, and assigns it the value (and type) of the operand on the right.
+= , -= , *= , /= , %=	Apply the indicated operation using the left and the right operator and store the result in the left operator. For instance, <code>a *= 8</code> implies <code>a = a * 8</code> .

Bitwise operators

Go includes full support for manipulating values at their most elemental forms. The following summarizes bitwise operators supported by Go:

Operator	Description
&	Bitwise AND
	Bitwise OR
a ^ b	Bitwise XOR

& ^	Bitwise AND NOT
^a	Unary bitwise complement
<<	Left-shift
>>	Right-shift

The right operand, in a shift operation, must be an unsigned integer or be able to be converted to an unsigned value. When the left operand is an untyped constant value, the compiler must be able to derive a signed integer type from its value or it will fail compilation.

The shift operators in Go also support both arithmetic and logical shifts. If the left operand is unsigned, Go automatically applies logical shift, whereas if it is signed, Go will apply an arithmetic shift.

Logical Operators

The following is a list of Go logical operations on Boolean values:

Operator	Operation
&&	Logical AND
	Logical OR
!	Logical NOT

Comparison operators

All Go types can be tested for equality, including basic and composite types. However, only string, integer, and floating-point values can be compared using ordering operators, as is summarized in the following table:

Operator	Operation	Supported type
==	Equal	String, numeric, Boolean, interface, pointer, and struct types
!=	Not Equal	String, numeric, Boolean, interface, pointer, and struct types

< , <= , > , >=	Ordering operators	String, integers, and floating points
--------------------------	--------------------	---------------------------------------

Operator precedence

Since Go has fewer operators than are found in its counterparts such as C or Java, its operator precedence rules are far simpler. The following table lists Go's operator precedence echelon, starting with the highest:

Operation	Precedence
Multiplicative	* , / , % , << , >> , & , &^
Additive	+ , - , , ^
Comparative	== , != , < , <= , > , >=
Logical AND	&&
Logical OR	

Summary

This chapter covered a lot of ground around the basic constructs of the Go language. It started with the structure of Go's source code text file and progressed to cover variable identifiers, declarations, and initializations. The chapter also provided extensive coverage of Go constants, constant declaration, and operators.

At this point, you may feel a bit overwhelmed by so much pedestrian information about the language and its syntax. The good news is that you don't have to know all of these details to be productive with the language. In the following chapters, we will continue to explore some of the more interesting bits about Go, including data types, functions, and packages.

3

Go Control Flow

Go borrows several of its control flow syntax from the C-family of languages. It supports all of the expected control structures, including `if...else`, `switch`, `for` loop, and even `goto`. Conspicuously absent, though, are `while` or `do...while` statements. The following topics in this chapter examine Go's control flow elements, some of which you may already be familiar with, and others that bring a new set of functionalities not found in other languages:

- The `if` statement
- The `switch` statement
- The type `Switch`
- The `for` statement

The `if` statement

The `if` statement, in Go, borrows its basic structural form from other C-like languages. The statement conditionally executes a code block when the Boolean expression that follows the `if` keyword evaluates to `true`, as illustrated in the following abbreviated program, which displays information about world currencies:

```
import "fmt"

type Currency struct {
    Name      string
    Country   string
    Number    int
}

var CAD = Currency{
```

```
Name: "Canadian Dollar",
Country: "Canada",
Number: 124}

var FJD = Currency{
    Name: "Fiji Dollar",
    Country: "Fiji",
    Number: 242}

var JMD = Currency{
    Name: "Jamaican Dollar",
    Country: "Jamaica",
    Number: 388}

var USD = Currency{
    Name: "US Dollar",
    Country: "USA",
    Number: 840}

func main() {
    num0 := 242
    if num0 > 100 || num0 < 900 {
        fmt.Println("Currency: ", num0)
        printCurr(num0)
    } else {
        fmt.Println("Currency unknown")
    }

    if num1 := 388; num1 > 100 || num1 < 900 {
        fmt.Println("Currency:", num1)
        printCurr(num1)
    }
}

func printCurr(number int) {
    if CAD.Number == number {
        fmt.Printf("Found: %v\n", CAD)
    } else if FJD.Number == number {
        fmt.Printf("Found: %v\n", FJD)
    } else if JMD.Number == number {
        fmt.Printf("Found: %v\n", JMD)
    } else if USD.Number == number {
        fmt.Printf("Found: %v\n", USD)
    } else {
        fmt.Println("No currency found with number", number)
    }
}
```

golang.fyi/ch03/ifstmt.go

The `if` statement in Go looks similar to other languages. However, it sheds a few syntactic rules, while enforcing new ones:

- The parentheses around the test expression are not necessary. While the following `if` statement will compile, it is not idiomatic:

```
if (num0 > 100 || num0 < 900) {  
    fmt.Println("Currency: ", num0)  
    printCurr(num0)  
}
```

- Use the following instead:

```
if num0 > 100 || num0 < 900 {  
    fmt.Println("Currency: ", num0)  
    printCurr(num0)  
}
```

- The curly braces for the code block are always required. The following snippet will not compile:

```
if num0 > 100 || num0 < 900 printCurr(num0)
```

- However, this will compile:

```
if num0 > 100 || num0 < 900 {printCurr(num0)}
```

- It is idiomatic, however, to write the `if` statement on multiple lines (no matter how simple the statement block may be). This encourages good style and clarity. The following snippet will compile with no issues:

```
if num0 > 100 || num0 < 900 {printCurr(num0)}
```

- However, the preferred idiomatic layout for the statement is to use multiple lines, as follows:

```
if num0 > 100 || num0 < 900 {  
    printCurr(num0)  
}
```

- The `if` statement may include an optional `else` block, which is executed when the expression in the `if` block evaluates to `false`. The code in the `else` block must be wrapped in curly braces using multiple lines, as shown in the following snippet:

```
if num0 > 100 || num0 < 900 {  
    fmt.Println("Currency: ", num0)  
    printCurr(num0)  
} else {  
    fmt.Println("Currency unknown")  
}
```

- The `else` keyword may be immediately followed by another `if` statement forming an `if...else...if` chain, as used in the function `printCurr()` from the source code listed earlier:

```
if CAD.Number == number {  
    fmt.Printf("Found: %v\n", CAD)  
} else if FJD.Number == number {  
    fmt.Printf("Found: %v\n", FJD)  
}
```

The `if...else...if` statement chain can grow as long as needed and may be terminated by an optional `else` statement to express all other untested conditions. Again, this is done in the `printCurr()` function, which tests four conditions using the `if...else...if` blocks. Lastly, it includes an `else` statement block to catch any other untested conditions:

```
func printCurr(number int) {  
    if CAD.Number == number {  
        fmt.Printf("Found: %v\n", CAD)  
    } else if FJD.Number == number {  
        fmt.Printf("Found: %v\n", FJD)  
    } else if JMD.Number == number {  
        fmt.Printf("Found: %v\n", JMD)  
    } else if USD.Number == number {  
        fmt.Printf("Found: %v\n", USD)  
    } else {  
        fmt.Println("No currency found with number", number)  
    }  
}
```

In Go, however, the idiomatic, and cleaner, way to write such a deep `if...else...if` code block is to use an expressionless `switch` statement. This is covered later, in the *Switch statement* section.

The if statement initialization

The `if` statement supports a composite syntax where the tested expression is preceded by an initialization statement. At runtime, the initialization is executed before the test expression is evaluated, as illustrated in this code snippet (from the program listed earlier):

```
if num1 := 388; num1 > 100 || num1 < 900 {  
    fmt.Println("Currency:", num1)  
    printCurr(num1)  
}
```

The initialization statement follows normal variable declaration and initialization rules. The scope of the initialized variables is bound to the `if` statement block, beyond which they become unreachable. This is a commonly used idiom in Go and is supported in other flow control constructs covered in this chapter.

Switch statements

Go also supports a `switch` statement similar to that found in other languages such as, C or Java. The `switch` statement in Go achieves multi-way branching by evaluating values or expressions from `case` clauses, as shown in the following, abbreviated, source code:

```
import "fmt"  
  
type Curr struct {  
    Currency string  
    Name      string  
    Country   string  
    Number    int  
}  
  
var currencies = []Curr{  
    Curr{"DZD", "Algerian Dinar", "Algeria", 12},  
    Curr{"AUD", "Australian Dollar", "Australia", 36},  
    Curr{"EUR", "Euro", "Belgium", 978},  
    Curr{"CLP", "Chilean Peso", "Chile", 152},  
    Curr{"EUR", "Euro", "Greece", 978},  
    Curr{"HTG", "Gourde", "Haiti", 332},  
    ...  
}  
  
func isDollar(curr Curr) bool {  
    var bool result  
    switch curr {
```

```
default:
    result = false
case Curr{"AUD", "Australian Dollar", "Australia", 36}:
    result = true
case Curr{"HKD", "Hong Kong Dollar", "Hong Koong", 344}:
    result = true
case Curr{"USD", "US Dollar", "United States", 840}:
    result = true
}
return result
}
func isDollar2(curr Curr) bool {
    dollars := []Curr{currencies[2], currencies[6], currencies[9]}
    switch curr {
    default:
        return false
    case dollars[0]:
        fallthrough
    case dollars[1]:
        fallthrough
    case dollars[2]:
        return true
    }
    return false
}
func isEuro(curr Curr) bool {
    switch curr {
    case currencies[2], currencies[4], currencies[10]:
        return true
    default:
        return false
    }
}
func main() {
    curr := Curr{"EUR", "Euro", "Italy", 978}
    if isDollar(curr) {
        fmt.Printf("%+v is Dollar currency\n", curr)
    } else if isEuro(curr) {
        fmt.Printf("%+v is Euro currency\n", curr)
    } else {
        fmt.Println("Currency is not Dollar or Euro")
    }
    dol := Curr{"HKD", "Hong Kong Dollar", "Hong Koong", 344}
    if isDollar2(dol) {
        fmt.Println("Dollar currency found:", dol)
    }
}
```

```
}
```

golang.fyi/ch03/switchstmt.go

The `switch` statement in Go has some interesting properties and rules that make it easy to use and reason about:

- Semantically, Go's `switch` statement can be used in two contexts:
 - An expression `switch` statement
 - A type `switch` statement
- The `break` statement can be used to escape out of a `switch` code block early.
- The `switch` statement can include a default case when no other case expressions evaluate to a match. There can only be one default case and it may be placed anywhere within the `switch` block.

Using expression switches

Expression switches are flexible and can be used in many contexts where control flow of a program needs to follow multiple path. An expression switch supports many attributes, as outlined in the following bullets:

- Expression switches can test values of any types. For instance, the following code snippet (from the previous program listing) tests variable `Curr` of type `struct`:

```
func isDollar(curr Curr) bool {
    var bool result
    switch curr {
        default:
            result = false
        case Curr{"AUD", "Australian Dollar", "Australia", 36}:
            result = true
        case Curr{"HKD", "Hong Kong Dollar", "Hong Koong", 344}:
            result = true
        case Curr{"USD", "US Dollar", "United States", 840}:
            result = true
    }
    return result
}
```

- The expressions in `case` clauses are evaluated from left to right, top to bottom, until a value (or expression) is found that is equal to that of the `switch` expression.
- Upon encountering the first case that matches the `switch` expression, the program will execute the statements for the `case` block and then immediately exit the `switch` block. Unlike other languages, the Go `case` statement does not need to use a `break` to avoid falling through the next case (see the *Fallthrough cases* section). For instance, calling `isDollar(Curr{"HKD", "Hong Kong Dollar", "Hong Kong", 344})` will match the second `case` statement in the preceding function. The code will set the result to `true` and exit the `switch` code block immediately.
- Case clauses can have multiple values (or expressions) separated by commas with a logical OR operator implied between them. For instance, in the following snippet, the `switch` expression `curr` is tested against values `currencies[2]`, `currencies[4]`, or `currencies[10]`, using one case clause until a match is found:

```
func isEuro(curr Curr) bool {
    switch curr {
        case currencies[2], currencies[4], currencies[10]:
            return true
        default:
            return false
    }
}
```

- The `switch` statement is the cleaner and preferred idiomatic approach to writing complex conditional statements in Go. This is evident when the preceding snippet is compared to the following, which does the same comparison using `if` statements:

```
func isEuro(curr Curr) bool {
    if curr == currencies[2] || curr == currencies[4],
       curr == currencies[10]{
        return true
    }else{
        return false
    }
}
```

The fallthrough cases

There is no automatic *fall through* in Go's `case` clause as there is in the C or Java `switch` statements. Recall that a `switch` block will exit after executing its first matching case. The code must explicitly place the `fallthrough` keyword, as the last statement in a `case` block, to force the execution flow to fall through the successive `case` block. The following code snippet shows a `switch` statement with a `fallthrough` in each `case` block:

```
func isDollar2(curr Curr) bool {
    switch curr {
    case Curr{"AUD", "Australian Dollar", "Australia", 36}:
        fallthrough
    case Curr{"HKD", "Hong Kong Dollar", "Hong Kong", 344}:
        fallthrough
    case Curr{"USD", "US Dollar", "United States", 840}:
        return true
    default:
        return false
    }
}
```

golang.fyi/ch03/switchstmt.go

When a case is matched, the `fallthrough` statements cascade down to the first statement of the successive `case` block. So, if `curr = Curr{"AUD", "Australian Dollar", "Australia", 36}`, the first case will be matched. Then the flow cascades down to the first statement of the second case block, which is also a `fallthrough` statement. This causes the first statement, to `return true`, of the third case block to execute. This is functionally equivalent to the following snippet:

```
switch curr {
case Curr{"AUD", "Australian Dollar", "Australia", 36},
    Curr{"HKD", "Hong Kong Dollar", "Hong Kong", 344},
    Curr{"USD", "US Dollar", "United States", 840}:
    return true
default:
    return false
}
```

Expressionless switches

Go supports a form of the `switch` statement that does not specify an expression. In this format, each `case` expression must evaluate to a Boolean value `true`. The following abbreviated source code illustrates the uses of an expressionless `switch` statement, as listed in function `find()`. The function loops through the slice of `Curr` values to search for a match based on field values in the `struct` function that's passed in:

```
import (
    "fmt"
    "strings"
)
type Curr struct {
    Currency string
    Name      string
    Country   string
    Number    int
}

var currencies = []Curr{
    Curr{"DZD", "Algerian Dinar", "Algeria", 12},
    Curr{"AUD", "Australian Dollar", "Australia", 36},
    Curr{"EUR", "Euro", "Belgium", 978},
    Curr{"CLP", "Chilean Peso", "Chile", 152},
    ...
}

func find(name string) {
    for i := 0; i < 10; i++ {
        c := currencies[i]
        switch {
        case strings.Contains(c.Currency, name),
            strings.Contains(c.Name, name),
            strings.Contains(c.Country, name):
            fmt.Println("Found", c)
        }
    }
}
```

golang.fyi/ch03/switchstmt2.go

Notice in the previous example, the `switch` statement in function `find()` does not include an expression. Each `case` expression is separated by a comma and must be evaluated to a Boolean value with an implied `OR` operator between each. The previous `switch` statement is equivalent to the following use of an `if` statement to achieve the same logic:

```
func find(name string) {
    for i := 0; i < 10; i++ {
        c := currencies[i]
        if strings.Contains(c.Currency, name) ||
            strings.Contains(c.Name, name) ||
            strings.Contains(c.Country, name) {
            fmt.Println("Found", c)
        }
    }
}
```

Switch initializer

The `switch` keyword may be immediately followed by a simple initialization statement where variables, local to the `switch` code block, may be declared and initialized. This convenient syntax uses a semi-colon between the initializer statement and the `switch` expression to declare variables, which may appear anywhere in the `switch` code block. The following code sample shows how this is done by initializing two variables, `name` and `curr`, as part of the `switch` declaration:

```
func assertEuro(c Curr) bool {
    switch name, curr := "Euro", "EUR"; {
        case c.Name == name:
            return true
        case c.Currency == curr:
            return true
    }
    return false
}
```

golang.fyi/ch03/switchstmt2.go

The previous code snippet uses an expressionless `switch` statement with an initializer. Notice the trailing semi-colon to indicate the separation between the initialization statement and the expression area for the switch. In the example, however, the `switch` expression is empty.

Type switches

Given Go's strong type support, it should be of little surprise that the language supports the ability to query type information. The type `switch` is a statement that uses the Go interface type to compare the underlying type information of values (or expressions). A full discussion on interface types and type assertion is beyond the scope of this section. You can find more details on the subject in [Chapter 8, Methods, Interfaces, and Objects](#).

Nevertheless, for the sake of completeness, a short discussion on type switches is provided here. For now, all you need to know is that Go offers the type `interface{}`, or empty interface, as a super type that is implemented by all other types in the type system. When a value is assigned type `interface{}`, it can be queried using the type `switch`, as shown in function `findAny()` in the following code snippet, to query information about its underlying type:

```
func find(name string) {
    for i := 0; i < 10; i++ {
        c := currencies[i]
        switch {
        case strings.Contains(c.Currency, name),
            strings.Contains(c.Name, name),
            strings.Contains(c.Country, name):
                fmt.Println("Found", c)
        }
    }
}

func findNumber(num int) {
    for _, curr := range currencies {
        if curr.Number == num {
            fmt.Println("Found", curr)
        }
    }
}

func findAny(val interface{}) {
    switch i := val.(type) {
    case int:
        findNumber(i)
```

```
case string:  
    find(i)  
default:  
    fmt.Printf("Unable to search with type %T\n", val)  
}  
}  
  
func main() {  
    findAny("Peso")  
    findAny(404)  
    findAny(978)  
    findAny(false)  
}
```

golang.fyi/ch03/switchstmt2.go

The function `findAny()` takes an `interface{}` as its parameter. The type `switch` is used to determine the underlying type and value of the variable `val` using the type assertion expression:

```
switch i := val.(type)
```

Notice the use of the keyword `type` in the preceding type assertion expression. Each case clause will be tested against the type information queried from `val.(type)`. Variable `i` will be assigned the actual value of the underlying type and is used to invoke a function with the respective value. The default block is invoked to guard against any unexpected type assigned to the parameter `val` parameter. Function `findAny` may then be invoked with values of diverse types, as shown in the following code snippet:

```
findAny("Peso")  
findAny(404)  
findAny(978)  
findAny(false)
```

The for statements

As a language related to the C-family, Go also supports `for` loop style control structures. However, as you may have come to expect by now, Go's `for` statements work interestingly differently and simply. The `for` statement in Go supports four distinct idioms, as summarized in the following table:

For Statement	Usage
For condition	Used to semantically replace <code>while</code> and <code>do...while</code> loops: <code>for x < 10 { ... }</code>
Infinite loop	The conditional expression may be omitted to create an infinite loop: <code>for { ... }</code>
Traditional	This is the traditional form of the C-family <code>for</code> loop with the initializer, test, and update clauses: <code>for x:=0; x < 10; x++ { ... }</code>
For range	Used to iterate over an expression representing a collection of items stored in an array, string (array of rune), slice, map, and channel: <code>for i, val := range values { ... }</code>

Notice, as with all other control statements in Go, the `for` statements do not use parentheses around their expressions. All statements for the loop code block must be enclosed within curly brackets or the compiler will produce an error.

For condition

The `for` condition uses a construct that is semantically equivalent to the `while` loop found in other languages. It uses the keyword `for`, followed by a Boolean expression that allows the loop to proceed as long as it is evaluated to true. The following abbreviated source listing shows an example of this form of the `for` loop:

```
type Curr struct {  
    Currency string
```

```
Name      string
Country  string
Number   int
}

var currencies = []Curr{
    Curr{"KES", "Kenyan Shilling", "Kenya", 404},
    Curr{"AUD", "Australian Dollar", "Australia", 36},
    ...
}

func listCurrs(howlong int) {
    i := 0
    for i < len(currencies) {
        fmt.Println(currencies[i])
        i++
    }
}
```

golang.fyi/ch03/forstmt.go

The `for` statement, in function `listCurrs()`, iterates as long as the conditional expression `i < len(currencies)` returns true. Care must be taken to ensure the value of `i` is updated with each iteration to avoid creating an accidental infinite loop.

Infinite loop

When the Boolean expression is omitted in the `for` statement, the loop runs indefinitely, as shown the following example:

```
for {
    // statements here
}
```

This is equivalent to the `for(;;)` or the `while (true)` found in other languages, such as C or Java.

The traditional for statement

Go also supports the traditional form of the `for` statement, which includes an initialization statement, a conditional expression, and an update statement, all separated by a semi-colon. This is the form of the statement that is traditionally found in other C-like languages. The following source snippet illustrates the use of a traditional for statement in the function `sortByNumber`:

```
type Curr struct {
    Currency string
    Name      string
    Country   string
    Number    int
}

var currencies = []Curr{
    Curr{"KES", "Kenyan Shilling", "Kenya", 404},
    Curr{"AUD", "Australian Dollar", "Australia", 36},
    ...
}

func sortByNumber() {
    N := len(currencies)
    for i := 0; i < N-1; i++ {
        currMin := i
        for k := i + 1; k < N; k++ {
            if currencies[k].Number < currencies[currMin].Number {
                currMin = k
            }
        }
        // swap
        if currMin != i {
            temp := currencies[i]
            currencies[i] = currencies[currMin]
            currencies[currMin] = temp
        }
    }
}
```

golang.fyi/ch03/forstmt.go

The previous example implements a selection sort that sorts the slice `currencies` by comparing the `Number` field of each `struct` value. The different sections of the `for` statement are highlighted using the following snippet of code (from the preceding function):

```
for k := i + 1; k < N; k++ {
    if currencies[k].Number < currencies[currMin].Number {
        currMin = k
    }
}
```

It turns out that the traditional `for` statement is a superset of the other forms of the loop discussed so far, as summarized in the following table:

For statement	Description
<code>k:=initialize() for ; k < 10; ++{ ... }</code>	The initialization statement is omitted. Variable <code>k</code> is initialized outside of the <code>for</code> statement. The idiomatic way, however, is to initialize your variables with the <code>for</code> statement.
<code>for k:=0; k < 10; { ... }</code>	The <code>update</code> statement (after the last semi-colon) is omitted here. The developer must provide update logic elsewhere or you risk creating an infinite loop.
<code>for ; k < 10; { ... }</code>	This is equivalent to the <code>for</code> condition form (discussed earlier) <code>for k < 10 { ... }</code> . Again, the variable <code>k</code> is expected to be declared prior to the loop. Care must be taken to update <code>k</code> or you risk creating an infinite loop.
<code>for k:=0; ;k++{ ... }</code>	Here, the conditional expression is omitted. As before, this evaluates the conditional to <code>true</code> , which will produce an infinite loop if proper termination logic is not introduced in the loop.
<code>for ; ;{ ... }</code>	This is equivalent to the form <code>for{ ... }</code> and produces an infinite loop.

The initialization and the update statements, in the `for` loop, are regular Go statements. As such, they can be used to initialize and update multiple variables, as is supported by Go. To illustrate this point, the next example initializes and updates two variables, `w1` and `w2`, at the same time in the statement clauses:

```
import (
    "fmt"
    "math/rand"
)

var list1 = []string{
```

```
"break", "lake", "go",
"right", "strong",
"kite", "hello"}
```

```
var list2 = []string{
    "fix", "river", "stop",
    "left", "weak", "flight",
    "bye"}
```

```
func main() {
    rand.Seed(31)
    for w1, w2 := nextPair();
        w1 != "go" && w2 != "stop";
        w1, w2 = nextPair() {
            fmt.Printf("Word Pair -> [%s, %s]\n", w1, w2)
        }
}
```

```
func nextPair() (w1, w2 string) {
    pos := rand.Intn(len(list1))
    return list1[pos], list2[pos]
}
```

golang.fyi/ch03/forstmt2.go

The initialization statements initialize variables `w1` and `w2` by calling the function `nextPair()`. The condition uses a compound logical expression that will keep the loop running as long as it is evaluated to true. Lastly, variables `w1` and `w2` are both updated with each iteration of the loop by calling `nextPair()`.

The for range

Lastly, the `for` statement supports one additional form that uses the keyword `range` to iterate over an expression that evaluates to an array, slice, map, string, or channel. The `for-range` loop has this generic form:

`for [<identifier-list> :=] range <expression> { ... }`

Depending on the type produced by the `range` expression, there can be up to two variables emitted by each iteration, as summarized in the following table:

Range Expression	Range Variables
Loop over array or slice: <code>for i, v := range []V{1,2,3} { ... }</code>	The range produces two values, where <code>i</code> is the loop index and <code>v</code> is the value <code>v[i]</code> from the collection. Further discussions on array and slice are covered in Chapter 7, <i>Composite Types</i> .
Loop over string value: <code>for i, v := range "Hello" { ... }</code>	The range produces two values, where <code>i</code> is the index of byte in the string and <code>v</code> is the value of the UTF-8 encoded byte at <code>v[i]</code> returned as a rune. Further discussion on the string type is covered in Chapter 4, <i>Data Types</i> .
Loop over map: <code>for k, v := range map[K]V { ... }</code>	The range produces two values, where <code>k</code> is assigned the value of the map key of type <code>K</code> and <code>v</code> gets stored at <code>map[k]</code> of type <code>V</code> . Further discussion on map is covered in Chapter 7, <i>Composite Types</i> .
Loop on channel values: <code>var ch chan T for c := range ch { ... }</code>	An adequate discussion of channels is covered in Chapter 9, <i>Concurrency</i> . A channel is a two-way conduit able to receive and emit values. The <code>for...range</code> statement assigns each value received from the channel to variable <code>c</code> with each iteration.

You should be aware that the value emitted with each iteration is a copy of the original item stored in the source. For instance, in the following program, the values in the slice do not get updated after the loop completes:

```
import "fmt"

func main() {
    vals := []int{4, 2, 6}
    for _, v := range vals {
        v--
    }
    fmt.Println(vals)
}
```

To update the original value using the `for...range` loop, use the index expression to access the original value, as illustrated in the following.

```
func main() {
    vals := []int{4, 2, 6}
    for i, v := range vals {
        vals[i] = v - 1
    }
    fmt.Println(vals)
}
```

In the previous example, value `i` is used in a slice index expression `vals[i]` to update the original value stored in the slice. It is possible to omit the iteration value (the second variable in the assignment) if you only need access to the index value of an array, slice, or string (or key for a map). For instance, in the following example, the `for...range` statement only emits the current index value with each iteration:

```
func printCurrencies() {
    for i := range currencies {
        fmt.Printf("%d: %v\n", i, currencies[i])
    }
}
```

golang.fyi/ch03/for-range-stmt.go

Finally, there are some situations where you may not be interested in any of the values generated by the iteration, but rather the iteration mechanic itself. The next form of the `for` statement was introduced (as of Version 1.4 of Go) to express a `for range` without any variable declaration as shown in the following code snippet:

```
func main() {
    for range []int{1,1,1,1} {
        fmt.Println("Looping")
    }
}
```

The previous code will print "Looping" four times on the standard output. This form of the `for...range` loop is used sometimes when the range expression is over a channel. It is used to simply notify of the presence of a value in the channel.

The break, continue, and goto statements

Go supports a group of statements designed specifically to exit abruptly out of a running code block, such as switch and for statement, and transfer control to a different section of the code. All three statements can accept a label identifier that specifies a targeted location in the code where control is to be transferred.

The label identifier

Before diving into the core of this section, it is worthwhile to look at the label used by these statements. Declaring a label in Go requires an identifier followed by a colon, as shown in the following snippet:

```
DoSearch:
```

Naming your label is a matter of style. However, one should follow the identifier naming guidelines covered in the previous chapter. A label must be enclosed within a function. The Go compiler will not allow unused labels to dangle in the code. Similar to variables, if a label is declared, it must be referenced in the code.

The break statement

As in other C-like languages, the Go `break` statement terminates and exits the innermost enclosing `switch` or `for` statement code block and transfers control to another part of the running program. The `break` statement can accept an optional label identifier specifying a labeled location, in the enclosing function, where the flow of the program will resume. Here are some attributes of the label for the `break` statement to remember:

- The label must be declared within the same running function where the `break` statement is located
- A declared label must be followed immediately by the enclosing control statement (a `for` loop or `switch` statement) where the `break` is nested

If a `break` statement is followed by a label, control is transferred, not to the location where the label is, but rather to the statement immediately following the labeled block. If a label is not provided, the `break` statement abruptly exits and transfers control to the next statement following its enclosing `for` statement (or `switch` statement) block.

The following code is an overly exaggerated linear search that illustrates the working of the `break` statement. It does a word search and exits once the first instance of the word is found in the slice:

```
import (
    "fmt"
)

var words = [][]string{
    {"break", "lake", "go", "right", "strong", "kite", "hello"},  

    {"fix", "river", "stop", "left", "weak", "flight", "bye"},  

    {"fix", "lake", "slow", "middle", "sturdy", "high", "hello"},  

}

func search(w string) {  

    DoSearch:  

        for i := 0; i < len(words); i++ {  

            for k := 0; k < len(words[i]); k++ {  

                if words[i][k] == w {  

                    fmt.Println("Found", w)  

                    break DoSearch  

                }
            }
        }
}
```

`golang.fyi/ch03/breakstmt.go`

In the previous code snippet, the `break DoSearch` statement will essentially exit out of the innermost `for` loop and cause the execution flow to continue after the outermost labeled `for` statement, which in this example, will simply end the program.

The `continue` statement

The `continue` statement causes the control flow to immediately terminate the current iteration of the enclosing `for` loop and jump to the next iteration. The `continue` statement can take an optional label as well. The label has similar properties to that of the `break` statement:

- The label must be declared within the same running function where the `continue` statement is located
- The declared label must be followed immediately by an enclosing `for` loop statement where the `continue` statement is nested

When present, the `continue` statement is reached within a `for` statement block, the `for` loop will be abruptly terminated and control will be transferred to the outermost labeled `for` loop block for continuation. If a label is not specified, the `continue` statement will simply transfer control to the start of its enclosing `for` loop block for continuation of the next iteration.

To illustrate, let us revisit the previous example of word search. This version uses a `continue` statement, which causes the search to find multiple occurrences of the searched word in the slice:

```
func search(w string) {
    DoSearch:
        for i := 0; i < len(words); i++ {
            for k := 0; k < len(words[i]); k++ {
                if words[i][k] == w {
                    fmt.Println("Found", w)
                    continue DoSearch
                }
            }
        }
}
```

golang.fyi/ch03/breakstmt2.go

The `continue DoSearch` statement causes the current iteration of the innermost loop to stop and transfer control to the labeled outer loop, causing it to continue with the next iteration.

The `goto` statement

The `goto` statement is more flexible, in that it allows flow control to be transferred to an arbitrary location, inside a function, where a target label is defined. The `goto` statement causes an abrupt transfer of control to the label referenced by the `goto` statement. The following shows Go's `goto` statement in action in a simple, but functional example:

```
import "fmt"

func main() {
    var a string
Start:
    for {
        switch {
        case a < "aaa":
            goto A
    }
}

A:
```

```
case a >= "aaa" && a < "aaabbb":  
    goto B  
case a == "aaabbb":  
    break Start  
}  
A:  
a += "a"  
continue Start  
B:  
a += "b"  
continue Start  
}  
fmt.Println(a)  
}
```

golang.fyi/ch03/gotostmt.go

The code uses the `goto` statement to jump to different sections of the `main()` function. Notice that the `goto` statement can target labels defined anywhere in the code. The superfluous usage of the `Start:` label is left in the code for completeness and is not necessary in this context (since `continue`, without the label, would have the same effect). The following provides some guidance when using the `goto` statement:

- Avoid using the `goto` statement unless the logic being implemented can only be achieved using `goto` branching. This is because overuse of the `goto` statement can make code harder to reason about and debug.
- Place `goto` statements and their targeted label within the same enclosing code block when possible.
- Avoid placing labels where a `goto` statement will cause the flow to skip new variable declarations or cause them to be re-declared.
- `Go` will let you jump from inner to outer enclosing code blocks.
- It is a compilation error if you try to jump to a peer or to an enclosing code block.

Summary

This chapter provided a walkthrough of the mechanism of control flow in Go, including `if`, `switch`, and `for` statements. While Go's flow control constructs appear simple and easy to use, they are powerful and implement all branching primitives expected of a modern language. Readers are introduced to each concept with ample detail and examples to ensure clarity of the topics. The next chapter continues our look into Go fundamentals by introducing the reader to the Go type systems.

4

Data Types

Go is a strongly-typed language, which means any language element that stores (or expression that produces) a value has a type associated with it. In this chapter, readers will learn about the features of the type system as they explore the common data types supported by the language as outlined in the following:

- Go types
- Numeric types
- Boolean type
- Pointers
- Type declaration
- Type conversion

Go types

To help launch the conversation about types, let us take a peek at the types available. Go implements a simple type system that provides programmers direct control over how memory is allocated and laid out. When a program declares a variable, two things must take place:

- The variable must receive a type
- The variable will also be bound to a value (even when none is assigned)

This allows the type system to allocate the number of bytes necessary to store the declared value. The memory layout for declared variables maps directly to their declared types. There is no type boxing or automatic type conversion that takes place. The space you expect to be allocated is actually what gets reserved in memory.

To demonstrate this fact, the following program uses a special package called `unsafe` to circumvent the type system and extract memory size information for declared variables. It is important to note that this is purely illustrative as most programs do not commonly make use of the `unsafe` package.

```
package main
import (
    "fmt"
    "unsafe"
)

var (
    a uint8     = 72
    b int32    = 240
    c uint64   = 1234564321
    d float32  = 12432345.232
    e int64    = -1233453443434
    f float64  = -1.43555622362467
    g int16    = 32000
    h [5]rune = [5]rune{'O', 'n', 'T', 'o', 'p'}
)

func main() {
    fmt.Printf("a = %v [%T, %d bits]\n", a, a, unsafe.Sizeof(a)*8)
    fmt.Printf("b = %v [%T, %d bits]\n", b, b, unsafe.Sizeof(b)*8)
    fmt.Printf("c = %v [%T, %d bits]\n", c, c, unsafe.Sizeof(c)*8)
    fmt.Printf("d = %v [%T, %d bits]\n", d, d, unsafe.Sizeof(d)*8)
    fmt.Printf("e = %v [%T, %d bits]\n", e, e, unsafe.Sizeof(e)*8)
    fmt.Printf("f = %v [%T, %d bits]\n", f, f, unsafe.Sizeof(f)*8)
    fmt.Printf("g = %v [%T, %d bits]\n", g, g, unsafe.Sizeof(g)*8)
    fmt.Printf("h = %v [%T, %d bits]\n", h, h, unsafe.Sizeof(h)*8)
}
```

golang.fyi/ch04/alloc.go

When the program is executed, it prints out the amount of memory (in bits) consumed by each declared variable:

```
$>go run alloc.go
a = 72 [uint8, 8 bits]
b = 240 [int32, 32 bits]
c = 1234564321 [uint64, 64 bits]
d = 1.2432345e+07 [float32, 32 bits]
e = -1233453443434 [int64, 64 bits]
f = -1.43555622362467 [float64, 64 bits]
g = 32000 [int16, 16 bits]
h = [79 110 84 111 112] [[5]int32, 160 bits]
```

From the preceding output, we can see that variable `a` (of type `uint8`) will be stored using eight bits (or one byte), variable `b` using 32 bits (or four bytes), and so on. With the ability to influence memory consumption coupled with Go's support for pointer types, programmers are able to strongly control how memory is allocated and consumed in their programs.

This chapter will cover the types listed in the following table. They include basic types such as numeric, Boolean, and strings:

Type	Description
<code>string</code>	Type for storing text values
<code>rune</code>	An integer type (<code>int32</code>) used to represent characters.
<code>byte, int, int8, int16, int32, int64, rune, uint, uint8, uint16, uint32, uint64, uintptr</code>	Types for storing integral values.
<code>float32, float64</code>	Types for storing floating point decimal values.
<code>complex64, complex128</code>	Types that can represent complex numbers with both real and imaginary parts.
<code>bool</code>	Type for Boolean values.
<code>*T, pointer to type T</code>	A type that represents a memory address where a value of type <code>T</code> is stored.

The remaining types supported by Go, such as those listed in the following table, include composite, interface, function, and channels. They are covered later in chapters dedicated to their respective topics.

Type	Description
<code>Array [n] T</code>	An ordered collection of fixed size <code>n</code> of numerically indexed sequence of elements of a type <code>T</code> .
<code>Slice [] T</code>	A collection of unspecified size of numerically indexed sequence of elements of type <code>T</code> .
<code>struct { }</code>	A structure is a composite type composed of elements known as fields (think of an object).
<code>map [K] T</code>	An unordered sequence of elements of type <code>T</code> indexed by a key of arbitrary type <code>K</code> .
<code>interface{ }</code>	A named set of function declarations that define a set of operations that can be implemented by other types.

func (T) R	A type that represents all functions with a given parameter type T and return type R .
chan T	A type for an internal communication channel to send or receive values of type T .

Numeric types

Go's numeric types include support for integral and decimal values with a variety of sizes ranging from 8 to 64 bits. Each numeric type has its own layout in memory and is considered unique by the type system. As a way of enforcing this, and to avoid any sort of confusion when porting Go on different platforms, the name of a numeric type reflects its size requirement. For instance, type `int16` indicates an integer type that uses 16 bits for internal storage. This means that numeric values must be explicitly be converted when crossing type boundaries in assignments, expressions, and operations.

The following program is not all that functional, since all values are assigned to the blank identifier. However, it illustrates all of the numeric data types supported in Go.

```
package main
import (
    "math"
    "unsafe"
)

var _ int8 = 12
var _ int16 = -400
var _ int32 = 12022
var _ int64 = 1 << 33
var _ int = 3 + 1415

var _ uint8 = 18
var _ uint16 = 44
var _ uint32 = 133121
var i uint64 = 23113233
var _ uint = 7542
var _ byte = 255
var _ uintptr = unsafe.Sizeof(i)

var _ float32 = 0.5772156649
var _ float64 = math.Pi

var _ complex64 = 3.5 + 2i
var _ complex128 = -5.0i
```

```
func main() {
    fmt.Println("all types declared!")
}
```

golang.fyi/ch04/nums.go

Unsigned integer types

The following table lists all available types that can represent unsigned integers and their storage requirements in Go:

Type	Size	Description
uint8	Unsigned 8-bit	Range 0 - 255
uint16	Unsigned 16-bit	Range 0 - 65535
uint32	Unsigned 32-bit	Range 0 - 4294967295
uint64	Unsigned 64-bit	Range 0 - 18446744073709551615
uint	Implementation specific	A pre-declared type designed to represent either the 32 or 64-bit integers. As of version 1.x of Go, uint represents a 32-bit unsigned integer.
byte	Unsigned 8-bit	Alias for the unit8 type.
uintptr	Unsigned	An unsigned integer type designed to store pointers (memory addresses) for the underlying machine architecture.

Signed integer types

The following table lists all available types that can represent signed integers and their storage requirements in Go:

Type	Size	Description
int8	Signed 8-bit	Range -128 - 127
int16	Signed 16-bit	Range -32768 - 32767
int32	Signed 32-bit	Range -2147483648 - 2147483647
int64	Signed 64-bit	Range -9223372036854775808 - 9223372036854775807

int	Implementation specific	A pre-declared type designed to represent either the 32 or 64-bit integers. As of version 1.x of Go, int represents a 32-bit signed integer.
-----	-------------------------	--

Floating point types

Go supports the following types for representation of decimal values using IEEE standards:

Type	Size	Description
float32	Signed 32-bit	IEEE-754 standard representation of single precision floating point values.
float64	Signed 64-bit	IEEE-754 standard representation of double-precision floating point values.

Complex number types

Go also supports representation of complex numbers with both imaginary and real parts as shown by the following table:

Type	Size	Description
complex64	float32	Represents complex numbers with real and imaginary parts stored as float32 values.
complex128	float64	Represents complex numbers with real and imaginary parts stored as float64 values.

Numeric literals

Go supports the natural representation of integer values using a sequence of digits with a combination of a sign and decimal point (as seen in the previous example). Optionally, Go integer literals can also represent hexadecimal and octal numbers as illustrated in the following program:

```
package main
import "fmt"

func main() {
    vals := []int{
        1024,
```

```
    0x0FF1CE,
    0x8BADF00D,
    0xBEEF,
    0777,
}
for _, i := range vals {
    if i == 0xBEEF {
        fmt.Printf("Got %d\n", i)
        break
    }
}
}
```

golang.fyi/ch04/intslit.go

Hexadecimal values are prepended with the `0x` or `(0X)` prefix while octal values start with the number `0` as shown in the previous example. Floating point values can be represented using both decimal and exponential notations as shown in the following examples:

```
package main

import "fmt"

func main() {
    p := 3.1415926535
    e := .5772156649
    x := 7.2E-5
    y := 1.616199e-35
    z := .416833e32

    fmt.Println(p, e, x, y, z)
}
```

golang.fyi/ch04/floating.go

The previous program shows several representations of floating point literals in Go. Numbers can include an optional exponent portion indicated by `e` (or `E`) at the end of the number. For instance, `1.616199e-35` in the code represents numerical value 1.616199×10^{-35} . Lastly, Go supports literals for expressing complex numbers as shown in the following example:

```
package main
import "fmt"

func main() {
    a := -3.5 + 2i
    fmt.Printf("%v\n", a)
```

```
    fmt.Printf("%+g, %+g\n", real(a), imag(a))  
}
```

golang.fyi/ch04/complex.go

In the previous example, variable `a` is assigned a complex number with both a real and an imaginary part. The imaginary literal is a floating point number followed by the letter `i`. Notice that Go also offers two built-in functions, `real()` and `imag()`, to deconstruct complex numbers into their real and imaginary parts respectively.

Boolean type

In Go, Boolean binary values are stored using the `bool` type. Although a variable of type `bool` is stored as a 1-byte value, it is not, however, an alias for a numeric value. Go provides two pre-declared literals, `true` and `false`, to represent Boolean values as shown in the following example:

```
package main  
import "fmt"  
  
func main() {  
    var readyToGo bool = false  
    if !readyToGo {  
        fmt.Println("Come on")  
    } else {  
        fmt.Println("Let's go!")  
    }  
}
```

golang.fyi/ch04/bool.go

Rune and string types

In order to start our discussion about the `rune` and `string` types, some background context is in order. Go can treat character and string literal constants in its source code as Unicode. It is a global standard whose goal is to catalog symbols for known writing systems by assigning a numerical value (known as code point) to each character.

By default, Go inherently supports UTF-8 which is an efficient way of encoding and storing Unicode numerical values. That is all the background needed to continue with this subject. No further detail will be discussed as it is beyond the scope of this book.

The rune

So, what exactly does the `rune` type have to do with Unicode? The `rune` is an alias for the `int32` type. It is specifically intended to store Unicode integer values encoded as UTF-8. Let us take a look at some `rune` literals in the following program:

```
package main
import (
    "fmt"
)

var (
    bksp = '\b'
    tab = '\t'
    nwln = '\n'
    char1 = 'φ'
    char2 = '𩿱'
    char3 = '語'
    char4 = '\u0369'
    char5 = '\xFA'
    char6 = '\u045'
)

func main() {
    fmt.Println(bksp)
    fmt.Println(tab)
    fmt.Println(nwln)
    fmt.Println(char1)
    fmt.Println(char2)
    fmt.Println(char3)
    fmt.Println(char4)
    fmt.Println(char5)
    fmt.Println(char6)
}
```

golang.fyi/ch04/rune.go

Each variable in the previous program stores a Unicode character as a `rune` value. In Go, the `rune` may be specified as a string literal constant surrounded by single quotes. The literal may be one of the following:

- A printable character (as shown with variables `char1`, `char2`, and `char3`)
- A single character escaped with backslash for non-printable control values as tab, linefeed, newline, and so on
- `\u` followed by Unicode values directly (`\u0369`)
- `\x` followed by two hex digits
- A backslash followed by three octal digits (`\045`)

Regardless of the `rune` literal value within the single quotes, the compiler compiles and assigns an integer value as shown by the printout of the previous variables:

```
$>go run runes.go
8
9
10
632
2438
35486
873
250
37
```

The string

In Go, a string is implemented as a slice of immutable byte values. Once a string value is assigned to a variable, the value of that string is never changed. Typically, string values are represented as constant literals enclosed within double quotes as shown in the following example:

```
package main
import "fmt"

var (
    txt  = "水 and 火"
    txt2 = "\u6C34\x20brings\x20\x6c\x69\x66\x65."
    txt3 =
        \u6C34\x20
        brings\x20
        \x6c\x69\x66\x65.
)

func main() {
    fmt.Printf("%s (%d)\n", txt, len(txt))
    for i := 0; i < len(txt); i++ {
        fmt.Printf("%U ", txt[i])
    }
    fmt.Println()
    fmt.Println(txt2)
    fmt.Println(txt3)
}
```

golang.fyi/ch04/string.go

The previous snippet shows variable `txt` being assigned a string literal containing seven characters including two embedded Chinese characters. As referenced earlier, the Go compiler will automatically interpret string literal values as Unicode characters and encode them using UTF-8. This means that under the cover, each literal character is stored as a rune and may end up taking more than one byte for storage per visible character. In fact, when the program is executed, it prints the length of `txt` as 11, instead of the expected seven characters for the string, accounting for the additional bytes used for the Chinese symbols.

Interpreted and raw string literals

The following snippet (from the previous example) includes two string literals assigned to variable `txt2` and `txt3` respectively. As you can see, these two literals have the exact same content, however, the compiler will treat them differently:

```
var (
    txt2 = "\u6C34\x20brings\x20\x6c\x69\x66\x65."
    txt3 =
        \u6C34\x20
        brings\x20
```

```
\x6c\x69\x66\x65.  
)  
  
golang.fyi/ch04/string.go
```

The literal value assigned to variable `txt2` is enclosed in double quotes. This is known as an interpreted string. An interpreted string may contain normal printable characters as well as backslash-escaped values which are parsed and interpreted as rune literals. So, when `txt2` is printed, the escape values are translated as the following string:

水 brings life.

Each symbol, in the interpreted string, corresponds to an escape value or a printable symbol as summarized in the following table:

水	<space>	brings	<space>	life	.
\u6C34	\x20	brings	\x20	\x6c\x69\x66\x65	.

On the other hand, the literal value assigned to variable `txt3` is surrounded by the grave accent characters ` `. This creates what is known as a raw string in Go. Raw string values are uninterpreted where escape sequences are ignored and all valid characters are encoded as they appear in the literal.

When variable `txt3` is printed, it produces the following output:

```
\u6C34\x20brings\x20\x6c\x69\x66\x65.
```

Notice that the printed string contains all the backslash-escaped values as they appear in the original string literal. Uninterpreted string literals are a great way to embed large multi-line textual content within the body of a source code without breaking its syntax.

Pointers

In Go, when a piece of data is stored in memory, the value for that data may be accessed directly or a pointer may be used to reference the memory address where the data is located. As with other C-family languages, pointers in Go provide a level of indirection that let programmers process data more efficiently without having to copy the actual data value every time it is needed.

Unlike C, however, the Go runtime maintains control of the management of pointers at runtime. A programmer cannot add an arbitrary integer value to the pointer to generate a new pointer address (a practice known as pointer arithmetic). Once an area of memory is referenced by a pointer, the data in that area will remain reachable until it is no longer referenced by any pointer variable. At that point, the unreferenced value becomes eligible for garbage collection.

The pointer type

Similar to C/C++, Go uses the `*` operator to designate a type as a pointer. The following snippet shows several pointers with different underlying types:

```
package main
import "fmt"

var valPtr *float32
var countPtr *int
var person *struct {
    name string
    age   int
}
var matrix *[1024]int
var row []*int64

func main() {
    fmt.Println(valPtr, countPtr, person, matrix, row)
}
```

golang.fyi/ch04/pointers.go

Given a variable of type `T`, Go uses expression `*T` as its pointer type. The type system considers `T` and `*T` as distinct and are not fungible. The zero value of a pointer, when it is not pointing to anything, is the address 0, represented by the literal constant `nil`.

The address operator

Pointer values can only be assigned addresses of their declared types. One way you can do so in Go is to use the address operator `&`(ampersand) to obtain the address value of a variable as shown in the following example:

```
package main
import "fmt"
```

```
func main() {
    var a int = 1024
    var aptr *int = &a

    fmt.Printf("a=%v\n", a)
    fmt.Printf("aptr=%v\n", aptr)
}
```

golang.fyi/ch04/pointers.go

Variable `aptr`, of pointer type `*int`, is initialized and assigned the address value of variable `a` using expression `&a` as listed here:

```
var a int = 1024
var aptr *int = &a
```

While variable `a` stores the actual value, we say that `aptr` points to `a`. The following shows the output of the program with the value of variable `a` and its memory location assigned to `aptr`:

```
a=1024
aptr=0xc208000150
```

The assigned address value will always be the same (always pointing to `a`) regardless of where `aptr` may be accessed in the code. It is also worth noting that Go does not allow the use of the address operator with literal constant for numeric, string, and bool types. Therefore, the following will not compile:

```
var aptr *int = &1024
fmt.Printf("a ptr1 = %v\n", aptr)
```

There is a syntactical exception to this rule, however, when initializing composite types such as struct and array with literal constants. The following program illustrates such scenarios:

```
package main
import "fmt"

func main() {
    structPtr := &struct{ x, y int }{44, 55}
    pairPtr := &[2]string{"A", "B"}

    fmt.Printf("struct=%#v, type=%T\n", structPtr, structPtr)
    fmt.Printf("pairPtr=%#v, type=%T\n", pairPtr, pairPtr)
}
```

golang.fyi/ch04/address2.go

In the previous code snippet, the address operator is used directly with composite literal `&struct{ x, y int }{44, 55}` and `&[2]string{"A", "B"}` to return pointer types `*struct { x int; y int }` and `*[2]string` respectively. This is a bit of syntactic sugar that eliminates the intermediary step of assigning the values to a variable, then retrieving their assigned addresses.

The new() function

The built-in function `new(<type>)` can also be used to initialize a pointer value. It first allocates the appropriate memory for a zero-value of the specified type. The function then returns the address for the newly created value. The following program uses the `new()` function to initialize variables `intptr` and `p`:

```
package main
import "fmt"

func main() {
    intptr := new(int)
    *intptr = 44

    p := new(struct{ first, last string })
    p.first = "Samuel"
    p.last = "Pierre"

    fmt.Printf("Value %d, type %T\n", *intptr, intptr)
    fmt.Printf("Person %+v\n", p)
}
```

golang.fyi/ch04/newptr.go

Variable `intptr` is initialized as `*int` and `p` as `*struct{first, last string}`. Once initialized, both values are updated accordingly later in the code. You can use the `new()` function to initialize pointer variables with zero values when the actual values are not available at the time of initialization.

Pointer indirection - accessing referenced values

If all you have is an address, you can access the value to which it points by applying the `*` operator to the pointer value itself (or dereferencing). The following program illustrates this idea in functions `double()` and `cap()`:

```
package main
import (
    "fmt"
    "strings"
)

func main() {
    a := 3
    double(&a)
    fmt.Println(a)
    p := &struct{ first, last string }{"Max", "Planck"}
    cap(p)
    fmt.Println(p)
}

func double(x *int) {
    *x = *x * 2
}

func cap(p *struct{ first, last string }) {
    p.first = strings.ToUpper(p.first)
    p.last = strings.ToUpper(p.last)
}
```

golang.fyi/ch04/derefptr.go

In the preceding code, the expression `*x = *x * 2`, in function `double()`, can be decomposed as follows to understand how it works:

Expression	Step
<code>*x * 2</code>	Original expression where <code>x</code> is of type <code>*int</code> .
<code>*(*x) * 2</code>	Dereferencing pointers by applying <code>*</code> to address values.
<code>3 * 2 = 6</code>	Dereferenced value of <code>*(*x) = 3</code> .
<code>*(*x) = 6</code>	The right side of this expression dereferences the value of <code>x</code> . It is updated with the result 6.

In function `cap()`, a similar approach is used to access and update fields in composite variable `p` of type `struct{first, last string}`. However, when dealing with composites, the idiom is more forgiving. It is not necessary to write `*p.first` to access the pointer's field value. We can drop the `*` and just use `p.first = strings.ToUpper(p.first)`.

Type declaration

In Go, it is possible to bind a type to an identifier to create a new named type that can be referenced and used wherever the type is needed. Declaring a type takes the general format as follows:

```
type <name identifier> <underlying type name>
```

The type declaration starts with the keyword `type` followed by a *name identifier* and the name of an existing *underlying type*. The underlying type can be a built-in named type such as one of the numeric types, a Boolean, or a string type as shown in the following snippet of type declarations:

```
type truth bool
type quart float64
type gallon float64
type node string
```



A type declaration can also use a composite *type literal* as its underlying type. Composite types include array, slice, map, and struct. This section focuses on non-composite types. For further details on composite types, refer to Chapter 7, *Composite Types*.

The following sample illustrates how named types work in their most basic forms. The code in the example converts temperature values. Each temperature unit is represented by a declared type including `fahrenheit`, `celsius`, and `kelvin`.

```
package main
import "fmt"

type fahrenheit float64
type celsius float64
type kelvin float64

func fahrToCel(f fahrenheit) celsius {
    return celsius((f - 32) * 5 / 9)
}
```

```
func fharToKel(f fahrenheit) celsius {
    return celsius((f-32)*5/9 + 273.15)
}

func celToFahr(c celsius) fahrenheit {
    return fahrenheit(c*5/9 + 32)
}

func celToKel(c celsius) kelvin {
    return kelvin(c + 273.15)
}

func main() {
    var c celsius = 32.0
    f := fahrenheit(122)
    fmt.Printf("%.2f \u00b0C = %.2f \u00b0K\n", c, celToKel(c))
    fmt.Printf("%.2f \u00b0F = %.2f \u00b0C\n", f, fharToCel(f))
}
```

golang.fyi/ch04/typedef.go

In the preceding code snippet, the new declared types are all based on the underlying built-in numeric type `float64`. Once the new type has been declared, it can be assigned to variables and participate in expressions just like its underlying type. The newly declared type will have the same zero-value and can be converted to and from its underlying type.

Type conversion

In general, Go considers each type to be different. This means under normal circumstances, values of different types are not fungible in assignment, function parameters, and expression contexts. This is true for built-in and declared types. For instance, the following will cause a build error due to type mismatch:

```
package main
import "fmt"

type signal int

func main() {
    var count int32
    var actual int
    var test int64 = actual + count

    var sig signal
    var event int = sig
```

```
    fmt.Println(test)
    fmt.Println(event)
}
```

golang.fyi/ch04/type_conv.go

The expression `actual + count` causes a build time error because both variables are of different types. Even though variables `actual` and `count` are of numeric types and `int32` and `int` have the same memory representation, the compiler still rejects the expression.

The same is true for declared named types and their underlying types. The compiler will reject assignment `var event int = sig` because type `signal` is considered to be different from type `int`. This is true even though `signal` uses `int` as its underlying type.

To cross type boundaries, Go supports a type conversion expression that converts value from one type to another. Type conversion is done using the following format:

`<target_type>(<value or expression>)`

The following code snippet fixes the previous example by converting the variables to the proper types:

```
type signal int
func main() {
    var count int32
    var actual int
    var test int32 = int32(actual) + count

    var sig signal
    var event int = int(sig)
}
```

golang.fyi/ch04/type_conv2.go

Note that in the previous snippet assignment expression `var test int32 = int32(actual) + count` converts variable `actual` to the proper type to match the rest of the expression. Similarly, expression `var event int = int(sig)` converts variable `sig` to match the target type `int` in the assignment.

The conversion expressions satisfy the assignment by explicitly changing the type of the enclosing values. Obviously, not all types can be converted from one to another. The following table summarizes common scenarios when type conversion is appropriate and allowed:

Description	Code
The target type and converted value are both simple numeric types.	<pre>var i int var i2 int32 = int32(i) var re float64 = float64(i + int(i2))</pre>
The target type and the converted value are both complex numeric types.	<pre>var cn64 complex64 var cn128 complex128 = complex128(cn64)</pre>
The target type and converted value have the same underlying types.	<pre>type signal int var sig signal var event int = int(sig)</pre>
The target type is a string and the converted value is a valid integer type.	<pre>a := string(72) b := string(int32(101)) c := string(rune(108))</pre>
The target type is string and the converted value is a slice of bytes, int32, or runes.	<pre>msg0 := string([]byte{'H', 'i'}) msg1 := string([]rune{'Y', 'o', 'u', '!'})</pre>
The target type is a slice of byte, int32, or rune values and the converted value is a string.	<pre>data0 := []byte("Hello") data0 := []int32("World!")</pre>

Additionally, the conversion rules also work when the target type and converted value are pointers that reference the same types. Besides these scenarios in the previous table, Go types cannot be explicitly converted. Any attempt to do so will result in a compilation error.

Summary

This chapter presented its readers with an introduction to the Go type system. The chapter opened with an overview of types and dove into a comprehensive exploration of the basic built-in types such as numeric, Boolean, string, and pointer types. The discussion continued by exposing the reader to other important topics such as named type definition. The chapter closed with coverage of the mechanics of type conversion. In coming chapters, you will get a chance to learn more about other types such as composite, function, and interface.

5

Functions in Go

One of Go's syntactical *tour de force* is via its support for higher-order functions as is found in dynamic languages such as Python or Ruby. As we will see in this chapter, a function is also a typed entity with a value that can be assigned to a variable. In this chapter, we are going to explore functions in Go covering the following topics:

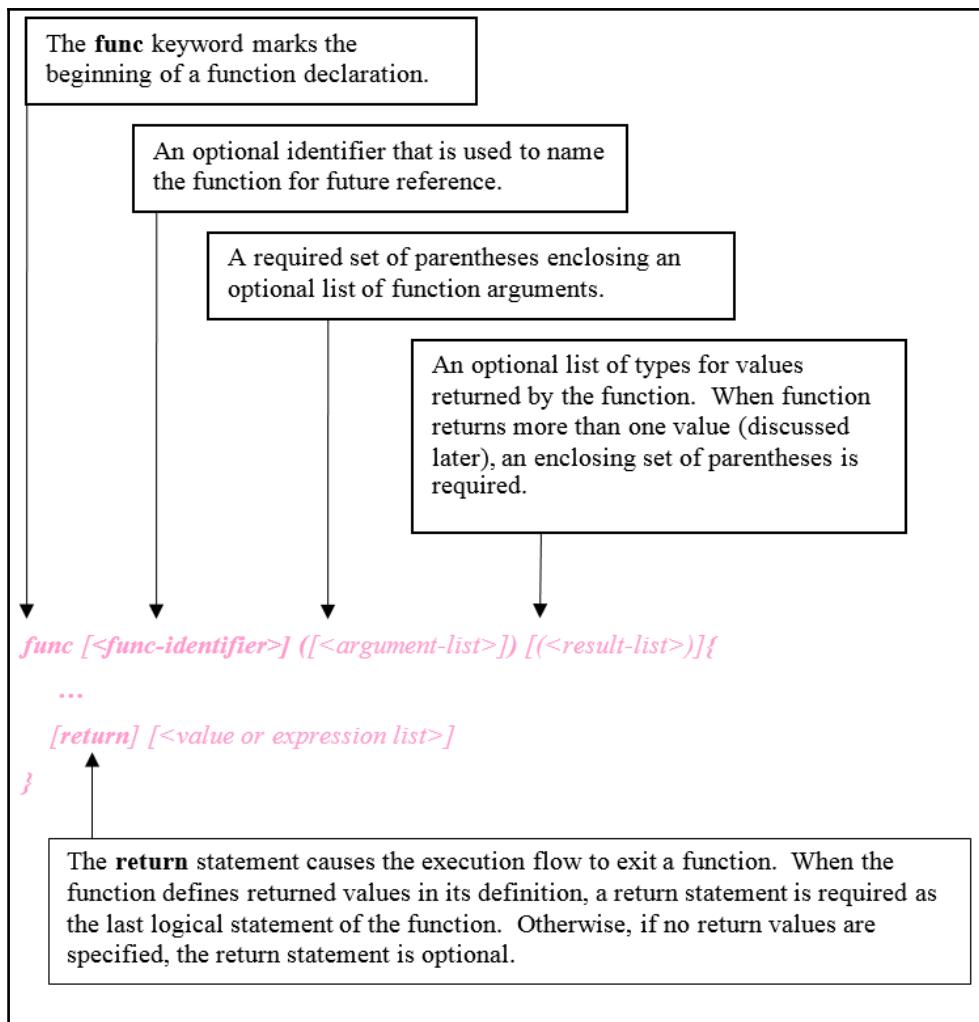
- Go functions
- Passing parameter values
- Anonymous functions and closures
- Higher-order functions
- Error signaling handling
- Deferring function calls
- Function panic and recovery

Go functions

In Go, functions are first-class, typed programming elements. A declared function literal always has a type and a value (the defined function itself) and can optionally be bound to a named identifier. Because functions can be used as data, they can be assigned to variables or passed around as parameters of other functions.

Function declaration

Declaring a function in Go takes the general form illustrated in the following figure. This canonical form is used to declare named and anonymous functions.



The most common form of function definition in Go includes the function's assigned identifier in the function literal. To illustrate this, the following table shows the source code of several programs with definitions of named functions with different combinations of parameters and return types.

Code	Description
<pre>package main import ("fmt" "math") func printPi() { fmt.Printf("printPi() %v\n", math.Pi) } func main() { printPi() }("fmt" "math") func printPi() { fmt.Printf("printPi() %v\n", math.Pi) } func main() { printPi() } golang.fyi/ch05/func0.go</pre>	A function with the name identifier <code>printPi</code> . It takes no parameter and returns no values. Notice when there is nothing to return, the <code>return</code> statement is optional.
<pre>package main import "fmt" func avogadro() float64 { return 6.02214129e23 } func main() { fmt.Printf("avogadro() = %e 1/mol\n", avogadro()) } golang.fyi/ch05/func1.go</pre>	A function named <code>avogadro</code> . It takes no parameter but returns a value of type <code>float64</code> . Notice the <code>return</code> statement is required when a return value is declared as part of the function's signature.

<pre>package main import "fmt" func fib(n int) { fmt.Printf("fib(%d):\n [", n) var p0, p1 uint64 = 0, 1 fmt.Printf("%d %d ", p0, p1) for i := 2; i <= n; i++ { p0, p1 = p1, p0+p1 fmt.Printf("%d ", p1) } fmt.Println("]") } func main() { fib(41) } golang.fyi/ch05/func2.go</pre>	<p>This defines the function <code>fib</code>. It takes parameter <code>n</code> of type <code>int</code> and prints the Fibonacci sequence for up to <code>n</code>. Again, nothing to return, therefore the <code>return</code> statement is omitted.</p>
<pre>package main import ("fmt" "math") func isPrime(n int) bool { lim := int(math.Sqrt(float64(n))) for p := 2; p <= lim; p++ { if (n % p) == 0 { return false } } return true } func main() { prime := 37 fmt.Printf("isPrime(%d) =\n%v\n", prime, isPrime(prime)) } golang.fyi/ch05/func3.go</pre>	<p>The last example defines the <code>isPrime</code> function. It takes a parameter of type <code>int</code> and returns a value of type <code>bool</code>. Since the function is declared to return a value of type <code>bool</code>, the last logical statement in the execution flow must be a <code>return</code> statement that returns a value of the declared type.</p>



Function signature The set of specified parameter types, result types, and the order in which those types are declared is known as the signature of the function. It is another unique characteristic that help identify a function. Two functions may have the same number of parameters and result values; however, if the order of those elements are different, then the functions have different signatures.

The function type

Normally, the name identifier, declared in a function literal, is used to invoke the function using an invocation expression whereby the function identifier is followed by a parameter list. This is what we have seen throughout the book so far and it is illustrated in the following example calling the `fib` function:

```
func main() {  
    fib(41)  
}
```

When, however, a function's identifier appears without parentheses, it is treated as a regular variable with a type and a value as shown in the following program:

```
package main  
import "fmt"  
  
func add(op0 int, op1 int) int {  
    return op0 + op1  
}  
  
func sub(op0, op1 int) int {  
    return op0 - op1  
}  
  
func main() {  
    var opAdd func(int, int) int = add  
    opSub := sub  
    fmt.Printf("op0(12,44)=%d\n", opAdd(12, 44))  
    fmt.Printf("sub(99,13)=%d\n", opSub(99, 13))  
}
```

golang.fyi/ch05/functype.go

The type of a function is determined by its signature. Functions are considered to be of the same type when they have the same number of arguments with the same types in the same order. In the previous example the `opAdd` variable is declared having the type `func (int, int) int`. This is the same signature as the declared functions `add` and `sub`. Therefore, the `opAdd` variable is assigned the `add` function variable. This allows `opAdd` to be invoked as you would invoke the `add` function.

The same is done for the `opSub` variable. It is assigned the value represented by the function identifier `sub` and type `func (int, int)`. Therefore, `opSub(99, 13)` invokes the second function, which returns the result of a subtraction.

Variadic parameters

The last parameter of a function can be declared as **variadic (variable length arguments)** by affixing ellipses (...) before the parameter's type. This indicates that zero or more values of that type may be passed to the function when it is called.

The following example implements two functions that accept variadic parameters. The first function calculates the average of the passed values and the second function sums up the numbers passed in as arguments:

```
package main
import "fmt"

func avg(nums ...float64) float64 {
    n := len(nums)
    t := 0.0
    for _, v := range nums {
        t += v
    }
    return t / float64(n)
}

func sum(nums ...float64) float64 {
    var sum float64
    for _, v := range nums {
        sum += v
    }
    return sum
}

func main() {
    fmt.Printf("avg([1, 2.5, 3.75]) =%.2f\n", avg(1, 2.5, 3.75))
    points := []float64{9, 4, 3.7, 7.1, 7.9, 9.2, 10}
```

```
    fmt.Printf("sum(%v) = %.2f\n", points, sum(points...))  
}
```

golang.fyi/ch05/funcvariadic.go

The compiler resolves the variadic parameter as a slice of type `[]float64` in both the preceding functions. The parameter values can then be accessed using a slice expression as shown in the previous example. To invoke functions with variadic arguments, simply provide a comma-separated list of values that matches the specified type as shown in the following snippet:

```
fmt.Printf("avg([1, 2.5, 3.75]) =%.2f\n", avg(1, 2.5, 3.75))
```

When no parameters are provided, the function receives an empty slice. The astute reader may be wondering, "Is it possible to pass in an existing slice of values as variadic arguments?" Thankfully, Go provides an easy idiom to handle such a case. Let's examine the call to the `sum` function in the following code snippet:

```
points := []float64{9, 4, 3.7, 7.1, 7.9, 9.2, 10}  
fmt.Printf("sum(%v) = %f\n", points, sum(points...))
```

A slice of floating-point values is declared and stored in variable `points`. The slice can be passed as a variadic parameter by adding ellipses to the parameter in the `sum(points...)` function call.

Function result parameters

Go functions can be defined to return one or more result values. So far in the book, most of the functions we have encountered have been defined to return a single result value. In general, a function is able to return a list of result values, with diverse types, separated by a comma (see the previous section, *Function declaration*).

To illustrate this concept, let us examine the following simple program which defines a function that implements an Euclidian division algorithm (see http://en.wikipedia.org/wiki/Division_algorithm). The `div` function returns both the quotient and the remainder values as its result:

```
package main  
import "fmt"  
  
func div(op0, op1 int) (int, int) {  
    r := op0  
    q := 0  
    for r >= op1 {
```

```
        q++
        r = r - op1
    }
    return q, r
}

func main() {
    q, r := div(71, 5)
    fmt.Printf("div(71,5) -> q = %d, r = %d\n", q, r)
}
```

golang.fyi/ch05/funcret0.go

The **return** keyword is followed by the number of result values matching (respectively) the declared results in the function's signature. In the previous example, the signature of the `div` function specifies two `int` values to be returned as result values. Internally, the function defines `int` variables `p` and `r` that are returned as result values upon completion of the function. Those returned values must match the types defined in the function's signature or risk compilation errors.

Functions with multiple result values must be invoked in the proper context:

- They must be assigned to a list of identifiers of the same types respectively
- They can only be included in expressions that expect the same number of returned values

This is illustrated in the following source snippet:

```
q, r := div(71, 5)
fmt.Printf("div(71,5) -> q = %d, r = %d\n", q, r)
```

Named result parameters

In general, the result list of a function's signature can be specified using variable identifiers along with their types. When using named identifiers, they are passed to the function as regular declared variables and can be accessed and modified as needed. Upon encountering a `return` statement, the last assigned result values are returned. This is illustrated in the following source snippet, which is a rewrite of the previous program:

```
func div(dvdn, dvsr int) (q, r int) {
    r = dvdn
    for r >= dvsr {
        q++
        r = r - dvsr
    }
}
```

```
    return  
}
```

golang.fyi/ch05/funcret1.go

Notice the `return` statement is naked; it omits all identifiers. As stated earlier, the values assigned in `q` and `r` will be returned to the caller. For readability, consistency, or style, you may elect not to use a naked `return` statement. It is perfectly legal to attach the identifier's name with the `return` statement (such as `return q, r`) as before.

Passing parameter values

In Go, all parameters passed to a function are done so by value. This means a local copy of the passed values is created inside the called function. There is no inherent concept of passing parameter values by reference. The following code illustrates this mechanism by modifying the value of the passed parameter, `val`, inside the `dbl` function:

```
package main  
import (  
    "fmt"  
    "math"  
)  
  
func dbl(val float64) {  
    val = 2 * val // update param  
    fmt.Printf("dbl()=% .5f\n", val)  
}  
  
func main() {  
    p := math.Pi  
    fmt.Printf("before dbl() p = % .5f\n", p)  
    dbl(p)  
    fmt.Printf("after dbl() p = % .5f\n", p)  
}
```

golang.fyi/ch05/funcpassbyval.go

When the program runs, it produces the following output that chronicles the state of the `p` variable before it is passed to the `dbl` function. The update is made locally to the passed parameter variable inside the `dbl` function, and lastly the value of the `p` variable after the `dbl` function is called:

```
$> go run funcpassbyval.go  
before dbl() p = 3.14159
```

```
dbl()=6.28319
after dbl() p = 3.14159
```

The preceding output shows that the original value assigned to variable `p` remains variable unchanged, even after it is passed to a function that seems to update its value internally. This is because the `val` parameter in the `dbl` function receives a local copy of the passed parameter.

Achieving pass-by-reference

While the pass-by-value is appropriate in many cases, it is important to note that Go can achieve pass-by-reference semantics using pointer parameter values. This allows a called function to reach outside of its lexical scope and change the value stored at the location referenced by the pointer parameter as is done in the `half` function in the following example:

```
package main
import "fmt"

func half(val *float64) {
    fmt.Printf("call half(%f)\n", *val)
    *val = *val / 2
}

func main() {
    num := 2.807770
    fmt.Printf("num=%f\n", num)
    half(&num)
    fmt.Printf("half(num)=%f\n", num)
}
```

golang.fyi/ch05/funcpassbyref.go

In the previous example, the call to the `half(&num)` function in `main()` updates, in place, the original value referenced by its `num` parameter. So, when the code is executed, it shows the original value of `num` and its value after the call to the `half` function:

```
$> go run funcpassbyref.go
num=2.807770
call half(2.807770)
half(num)=1.403885
```

As was stated earlier, Go function parameters are passed by value. This is true even when the function takes a pointer value as its parameter. Go still creates and passes in a local copy of the pointer value. In the previous example, the `half` function receives a copy of the pointer value it receives via the `val` parameter. The code uses pointer operator (*) to dereference and manipulate, in place, the value referenced by `val`. When the `half` function exits and goes out of scope, its changes are accessible by calling the `main` function.

Anonymous Functions and Closures

Functions can be written as literals without a named identifier. These are known as anonymous functions and can be assigned to a variable to be invoked later as shown in the following example:

```
package main
import "fmt"

var (
    mul = func(op0, op1 int) int {
        return op0 * op1
    }

    sqr = func(val int) int {
        return mul(val, val)
    }
)

func main() {
    fmt.Printf("mul(25,7) = %d\n", mul(25, 7))
    fmt.Printf("sqr(13) = %d\n", sqr(13))
}
```

golang.fyi/ch05/funcs.go

The previous program shows two anonymous functions declared and bound to the `mul` and `sqr` variables. In both cases, the functions take in parameters and return a value. Later in `main()`, the variables are used to invoke the function code bound to them.

Invoking anonymous function literals

It is worth noting that an anonymous function does not have to be bound to an identifier. The function literal can be evaluated, in place, as an expression that returns the function's result. This is done by ending the function literal with a list of argument values, enclosed in parentheses, as shown in the following program:

```
package main
import "fmt"

func main() {
    fmt.Printf(
        "94 °F) = %.2f (°C)\n",
        func(f float64) float64 {
            return (f - 32.0) * (5.0 / 9.0)
        }(94),
    )
}
```

golang.fyi/ch05/funcs.go

The literal format not only defines the anonymous function, but also invokes it. For instance, in the following snippet (from the previous program), the anonymous function literal is nested as a parameter to `fmt.Printf()`. The function itself is defined to accept a parameter and returns a value of type `float64`.

```
fmt.Printf(
    "94 °F) = %.2f (°C)\n",
    func(f float64) float64 {
        return (f - 32.0) * (5.0 / 9.0)
    }(94),
)
```

Since the function literal ends with a parameter list enclosed within parentheses, the function is invoked as an expression.

Closures

Go function literals are closures. This means they have lexical visibility to non-local variables declared outside of their enclosing code block. The following example illustrates this fact:

```
package main
import (
    "fmt"
```

```
"math"
)

func main() {
    for i := 0.0; i < 360.0; i += 45.0 {
        rad := func() float64 {
            return i * math.Pi / 180
        }()
        fmt.Printf("%.2f Deg = %.2f Rad\n", i, rad)
    }
}
```

github.com/vladimirvivien/learning-go/ch05/funcs.go

In the previous program, the function literal code block, `func() float64 {return deg * math.Pi / 180}()`, is defined as an expression that converts degrees to radians. With each iteration of the loop, a closure is formed between the enclosed function literal and the outer non-local variable, `i`. This provides a simpler idiom where the function naturally accesses non-local values without resorting to other means such as pointers.



In Go, lexically closed values can remain bounded to their closures long after the outer function that created the closure has gone out of scope. The garbage collector will handle cleanups as these closed values become unbounded.

Higher-order functions

We have already established that Go functions are values bound to a type. So, it should not be a surprise that a Go function can take another function as a parameter and also return a function as a result value. This describes the notion known as a higher-order function, which is a concept adopted from mathematics. While types such as `struct` let programmers abstract data, higher-order functions provide a mechanism to encapsulate and abstract behaviors that can be composed together to form more complex behaviors.

To make this concept clearer, let us examine the following program, which uses a higher-order function, `apply`, to do three things. It accepts a slice of integers and a function as parameters. It applies the specified function to each element in the slice. Lastly, the `apply` function also returns a function as its result:

```
package main
import "fmt"

func apply(nums []int, f func(int) int) func() {
```

```
    for i, v := range nums {
        nums[i] = f(v)
    }
    return func() {
        fmt.Println(nums)
    }
}

func main() {
    nums := []int{4, 32, 11, 77, 556, 3, 19, 88, 422}
    result := apply(nums, func(i int) int {
        return i / 2
    })
    result()
}
```

golang.fyi/ch05/funhighorder.go

In the program, the `apply` function is invoked with an anonymous function that halves each element in the slice as highlighted in the following snippet:

```
nums := []int{4, 32, 11, 77, 556, 3, 19, 88, 422}
result := apply(nums, func(i int) int {
    return i / 2
})
result()
```

As a higher-order function, `apply` abstracts the transformation logic which can be provided by any function of type `func(i int) int`, as shown next. Since the `apply` function returns a function, the variable `result` can be invoked as shown in the previous snippet.

As you explore this book, and the Go language, you will continue to encounter usage of higher-order functions. It is a popular idiom that is used heavily in the standard libraries. You will also find higher-order functions used in some concurrency patterns to distribute workloads (see Chapter 9, *Concurrency*).

Error signaling and handling

At this point, let us address how to idiomatically signal and handle errors when you make a function call. If you have worked with languages such as Python, Java, or C#, you may be familiar with interrupting the flow of your executing code by throwing an exception when an undesirable state arises.

As we will explore in this section, Go has a simplified approach to error signaling and error handling that puts the onus on the programmer to handle possible errors immediately after a called function returns. Go discourages the notion of interrupting an execution by indiscriminately short-circuiting the executing program with an exception in the hope that it will be properly handled further up the call stack. In Go, the traditional way of signaling errors is to return a value of type `error` when something goes wrong during the execution of your function. So let us take a closer look how this is done.

Signaling errors

To better understand what has been described in the previous paragraph, let us start with an example. The following source code implements an anagram program, as described in Column 2 from Jon Bentley's popular *Programming Pearls* book (second edition). The code reads a dictionary file (`dict.txt`) and groups all words with the same anagram. If the code does not quite make sense, please see `golang.fyi/ch05/anagram1.go` for an annotated explanation of how each part of the program works.

```
package main

import (
    "bufio"
    "bytes"
    "fmt"
    "os"
    "errors"
)

// sorts letters in a word (i.e. "morning" -> "gimnnor")
func sortRunes(str string) string {
    runes := bytes.Runes([]byte(str))
    var temp rune
    for i := 0; i < len(runes); i++ {
        for j := i + 1; j < len(runes); j++ {
            if runes[j] < runes[i] {
                temp = runes[i]
                runes[i], runes[j] = runes[j], temp
            }
        }
    }
    return string(runes)
}

// load loads content of file fname into memory as []string
```

```
func load(fname string) ([]string, error) {
    if fname == "" {
        return nil, errors.New(
            "Dictionary file name cannot be empty.")
    }

    file, err := os.Open(fname)
    if err != nil {
        return nil, err
    }
    defer file.Close()

    var lines []string
    scanner := bufio.NewScanner(file)
    scanner.Split(bufio.ScanLines)
    for scanner.Scan() {
        lines = append(lines, scanner.Text())
    }
    return lines, scanner.Err()
}

func main() {
    words, err := load("dict.txt")
    if err != nil {
        fmt.Println("Unable to load file:", err)
        os.Exit(1)
    }

    anagrams := make(map[string][]string)
    for _, word := range words {
        wordSig := sortRunes(word)
        anagrams[wordSig] = append(anagrams[wordSig], word)
    }

    for k, v := range anagrams {
        fmt.Println(k, "->", v)
    }
}
```

golang.fyi/ch05/anagram1.go

Again, if you want a more detail explanation of the previous program, take a look at the link supplied earlier. The focus here is on error signaling used in the previous program. As a convention, Go code uses the built-in type `error` to signal when an error occurred during execution of a function. Therefore, a function must return a value of type `error` to indicate to its caller that something went wrong. This is illustrated in the following snippet of the `load` function (extracted from the previous example):

```
func load(fname string) ([]string, error) {
    if fname == "" {
        return nil, errors.New(
            "Dictionary file name cannot be empty.")
    }

    file, err := os.Open(fname)
    if err != nil {
        return nil, err
    }
    ...
}
```

Notice that the `load` function returns multiple result parameters. One is for the expected value, in this case `[]string`, and the other is the error value. Idiomatic Go dictates that the programmer returns a non-nil value for result of type `error` to indicate that something abnormal occurred during the execution of the function. In the previous snippet, the `load` function signals an error occurrence to its callers in two possible instances:

- when the expected filename (`fname`) is empty
- when the call to `os.Open()` fails (for example, permission error, or otherwise)

In the first case, when a filename is not provided, the code returns an error using `errors.New()` to create a value of type `error` to exit the function. In the second case, the `os.Open` function returns a pointer representing the file and an error assigned to the `file` and `err` variables respectively. If `err` is not `nil` (meaning an error was generated), the execution of the `load` function is halted prematurely and the value of `err` is returned to be handled by the calling function further up the call stack.



When returning an error for a function with multiple result parameters, it is customary to return the zero-value for the other (non-error type) parameters. In the example, a value of `nil` is returned for the result of type `[]string`. While not necessary, it simplifies error handling and avoids any confusion for function callers.

Error handling

As described previously, signaling of an erroneous state is as simple as returning a non-nil value, of type `error`, during execution of a function. The caller may choose to handle the error or return it for further evaluation up the call stack as was done in the `load` function. This idiom forces errors to propagate upwards until they are handled at some point. The next snippet shows how the error generated by the `load` function is handled in the `main` function:

```
func main() {
    words, err := load("dict.txt")
    if err != nil {
        fmt.Println("Unable to load file:", err)
        os.Exit(1)
    }
    ...
}
```

Since the `main` function is the topmost caller in the call stack, it handles the error by terminating the entire program.

This is all there is to the mechanics of error handling in Go. The language forces the programmer to always test for an erroneous state on every function call that returns a value of the type `error`. The `if...not...nil` error handling idiom may seem excessive and verbose to some, especially if you are coming from a language with formal exception mechanisms. However, the gain here is that the program can construct a robust execution flow where programmers always know where errors may come from and handle them appropriately.

The error type

The `error` type is a built-in interface and, therefore must be implemented before it can be used. Fortunately, the Go standard library comes with implementations ready to be used. We have already used one of the implementations from the package, `errors`:

```
errors.New("Dictionary file name cannot be empty.")
```

You can also create parameterized error values using the `fmt.Errorf` function as shown in the following snippet:

```
func load(fname string) ([]string, error) {
    if fname == "" {
        return nil, errors.New(
            "Dictionary file name cannot be empty."
        )
    }
    // ...
}
```

```
        "Dictionary file name cannot be empty.")  
    }  
  
    file, err := os.Open(fname)  
    if err != nil {  
        return nil, fmt.Errorf(  
            "Unable to open file %s: %s", fname, err)  
    }  
    ...  
}
```

golang.fyi/ch05/anagram2.go

It is also idiomatic to assign error values to high-level variables so they can be reused throughout a program as needed. The following snippet pulled from <http://golang.org/src/os/error.go> shows the declaration of reusable errors associated with OS file operations:

```
var (  
    ErrInvalid     = errors.New("invalid argument")  
    ErrPermission  = errors.New("permission denied")  
    ErrExist       = errors.New("file already exists")  
    ErrNotExist    = errors.New("file does not exist")  
)
```

<http://golang.org/src/os/error.go>

You can also create your own implementation of the `error` interface to create custom errors. This topic is revisited in *Chapter 7, Methods, Interfaces, and Objects* where the book discusses the notion of extending types.

Deferring function calls

Go supports the notion of deferring a function call. Placing the keyword `defer` before a function call has the interesting effect of pushing the function unto an internal stack, delaying its execution right before the enclosing function returns. To better explain this, let us start with the following simple program that illustrates the use of `defer`:

```
package main  
import "fmt"  
  
func do(steps ...string) {  
    defer fmt.Println("All done!")  
    for _, s := range steps {  
        defer fmt.Println(s)
```

```
}

    fmt.Println("Starting")
}

func main() {
    do(
        "Find key",
        "Aplly break",
        "Put key in ignition",
        "Start car",
    )
}
```

golang.fyi/ch05/defer1.go

The previous example defines the `do` function that takes variadic parameter `steps`. The function defers the statement with `defer fmt.Println("All done!")`. Next, the function loops through slice `steps` and defers the output of each element with `defer fmt.Println(s)`. The last statement in the function `do` is a non-deferred call to `fmt.Println("Starting")`. Notice the order of the printed string values when the program is executed, as shown in the following output:

```
$> go run defer1.go
Starting
Start car
Put key in ignition
Aplly break
Find key
All done!
```

There are a couple facts that explain the reverse order of the printout. First, recall that deferred functions are executed right before their enclosing function returns. Therefore, the first value printed is generated by the last non-deferred method call. Next, as stated earlier, deferred statements are pushed into a stack. Therefore, deferred calls are executed using a last-in-first-out order. That is why "All done!" is the last string value printed in the output.

Using defer

The `defer` keyword modifies the execution flow of a program by delaying function calls. One idiomatic usage for this feature is to do a resource cleanup. Since `defer` will always get executed when the surrounding function returns, it is a good place to attach cleanup code such as:

- Closing open files
- Releasing network resources
- Closing the Go channel
- Committing database transactions
- And do on

To illustrate, let us return to our anagram example from earlier. The following code snippet shows a version of the code where `defer` is used to close the file after it has been loaded. The `load` function calls `file.Close()` right before it returns:

```
func load(fname string) ([]string, error) {  
    ...  
    file, err := os.Open(fname)  
    if err != nil {  
        return nil, err  
    }  
    defer file.Close()  
    ...  
}
```

golang.fyi/ch05/anagram2.go

The pattern of opening-defer-closing resources is widely used in Go. By placing the deferred intent immediately after opening or creating a resource allows the code to read naturally and reduces the likeliness of creating a resource leakage.

Function panic and recovery

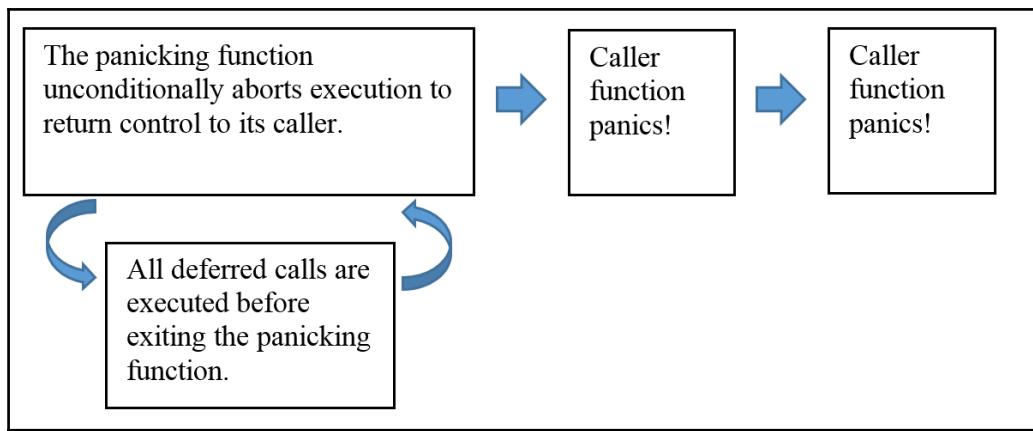
Earlier in the chapter, it was stated that Go does not have the traditional exception mechanism offered by other languages. Nevertheless, in Go, there is a way to abruptly exit an executing function known as function panic. Conversely, when a program is panicking, Go provides a way of recovering and regaining control of the execution flow.

Function panic

During execution, a function may panic because of any one of following:

- Explicitly calling the **panic** built-in function
- Using a source code package that panics due to an abnormal state
- Accessing a nil value or an out-of-bound array element
- Concurrency deadlock

When a function panics, it aborts and executes its deferred calls. Then its caller panics, causing a chain reaction as illustrated in the following figure:



The panic sequence continues all the way up the call stack until the `main` function is reached and the program exits (crashes). The following source code snippet shows a version of the anagram program that will cause an explicit panic if an output anagram file already exists when it tries to create one. This is done illustratively to cause the `write` function to panic when there is a file error:

```
package main
...
func write(fname string, anagrams map[string][]string) {
    file, err := os.OpenFile(
        fname,
        os.O_WRONLY+os.O_CREATE+os.O_EXCL,
        0644,
    )
    if err != nil {
        msg := fmt.Sprintf(
            "Unable to create output file: %v", err,
```

```
        )
    panic(msg)
}

...
}

func main() {
    words, err := load("dict.txt")
    if err != nil {
        fmt.Println("Unable to load file:", err)
        os.Exit(1)
    }
    anagrams := mapWords(words)
    write("out.txt", anagrams)
}
```

golang.fyi/ch05/anagram2.go

In the preceding snippet, the `write` function calls the `panic` function if `os.OpenFile()` method errors out. When the program calls the `main` function, if there is an output file already in the working directory, the program will panic and crash as shown in the following stack trace, indicating the sequence of calls that caused the crash:

```
> go run anagram2.go
panic: Unable to create output file: open out.txt: file exists
goroutine 1 [running]:
main.write(0x4e7b30, 0x7, 0xc2080382a0)
/Go/src/github.com/vladimirvivien/learning-go/ch05/anagram2.go:72 +0x1a3
main.main()
Go/src/github.com/vladimirvivien/learning-go/ch05/anagram2.go:103 +0x1e9
exit status 2
```

Function panic recovery

When a function panics, as explained earlier, it can crash an entire program. That may be the desired outcome depending on your requirements. It is possible, however, to regain control after a panic sequence has started. To do this, Go offers the built-in function called `recover`.

Recover works in tandem with panic. A call to function recover returns the value that was passed as an argument to panic. The following code shows how to recover from the panic call that was introduced in the previous example. In this version, the write function is moved inside makeAnagram() for clarity. When the write function is invoked from makeAnagram() and fails to open a file, it will panic. However, additional code is now added to recover:

```
package main
...
func write(fname string, anagrams map[string][]string) {
    file, err := os.OpenFile(
        fname,
        os.O_WRONLY+os.O_CREATE+os.O_EXCL,
        0644,
    )
    if err != nil {
        msg := fmt.Sprintf(
            "Unable to create output file: %v", err,
        )
        panic(msg)
    }
    ...
}

func makeAnagrams(words []string, fname string) {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Failed to make anagram:", r)
        }
    }()
    anagrams := mapWords(words)
    write(fname, anagrams)
}
func main() {
    words, err := load("")
    if err != nil {
        fmt.Println("Unable to load file:", err)
        os.Exit(1)
    }
    makeAnagrams(words, "")
}
```

golang.fyi/ch05/anagram3.go

To be able to recover from an unwinding panic sequence, the code must make a deferred call to the recover function. In the previous code, this is done in the `makeAnagrams` function by wrapping `recover()` inside an anonymous function literal, as highlighted in the following snippet:

```
defer func() {
    if r := recover(); r != nil {
        fmt.Println("Failed to make anagram:", r)
    }
}()
```

When the deferred `recover` function is executed, the program has an opportunity to regain control and prevent the panic from crashing the running program. If `recover()` returns `nil`, it means there is no current panic unwinding up the call stack or the panic was already handled downstream.

So, now when the program is executed, instead of crashing with a stack trace, the program recovers and gracefully displays the issue as shown in the following output:

```
> go run anagram3.go
Failed to make anagram: Unable to open output file for creation: open
out.txt: file exists
```



You may be wondering why we are using a `nil` to test the value returned by the `recover` function when a string was passed inside the call to `panic`. This is because both `panic` and `recover` take an empty interface type. As you will learn, the empty interface type is a generic type with the ability to represent any type in Go's type system. We will learn more about the empty interface in [Chapter 7, Methods, Interfaces and Objects](#) during discussions about interfaces.

Summary

This chapter presented its reader with an exploration of Go functions. It started with an overview of named function declarations, followed by a discussion on function parameters. The chapter delved into a discussion of function types and function values. The last portion of the chapter discussed the semantics of error handling, panic, and recovery. The next chapter continues the discussion of functions; however, it does so within the context of Go packages. It explains the role of a package as a logical grouping of Go functions (and other code elements) to form sharable and callable code modules.

6

Go Packages and Programs

Chapter 5, *Functions in Go* covered functions, the elementary level of abstraction for code organization that makes code addressable and reusable. This chapter continues up the ladder of abstraction with a discussion centered around Go packages. As will be covered in detail here, a package is a logical grouping of language elements stored in source code files that can be shared and reused, as covered in the following topics:

- The Go package
- Creating packages
- Building packages
- Package visibility
- Importing packages
- Package initialization
- Creating programs
- Remote packages

The Go package

Similar to other languages, Go source code files are grouped into compilable and sharable units known as packages. However, all Go source files must belong to a package (there is no such notion as a default package). This strict approach allows Go to keep its compilation rules and package resolution rules simple by favoring convention over configuration. Let us take a deep dive into the fundamentals of packages, their creation, use, and recommended practice.

Understanding the Go package

Before we dive into package creation and use, it is crucial to take a high-level view of the concept of packages to help steer the discussion later. A Go package is both a physical and a logical unit of code organization used to encapsulate related concepts that can be reused. By convention, a group of source files stored in the same directory are considered to be part of the same package. The following illustrates a simple directory tree, where each directory represents a package containing some source code:

```
foo
├── blat.go
└── bazz
    ├── quux.go
    └── qux.go
```

golang.fyi/ch06-foo

While not a requirement, it is a recommended convention to set a package's name, in each source file, to match the name of the directory where the file is located. For instance, source file `blat.go` is declared to be part of package `foo`, as shown in the following code, because it is stored in directory named `foo`:

```
package foo

import (
    "fmt"
    "foo/bar/bazz"
)

func fooIt() {
    fmt.Println("Foo!")
    bazz.Qux()
}
```

golang.fyi/ch06-foo/foo/blat.go

Files `quux.go` and `qux.go` are both part of package `bazz` since they are located in a directory with that name, as shown in the following code snippets:

```
package bazz
import "fmt"
func Qux() {
    fmt.Println("bazz.Quux")
}
golang.fyi/ch06-foo/foo/bazz/quux.go
```

```
package bazz
import "fmt"
func Quux() {
    Qux() fmt.Println("gazz.Quux")
}
golang.fyi/ch06-foo/foo/bazz/qux.go
```

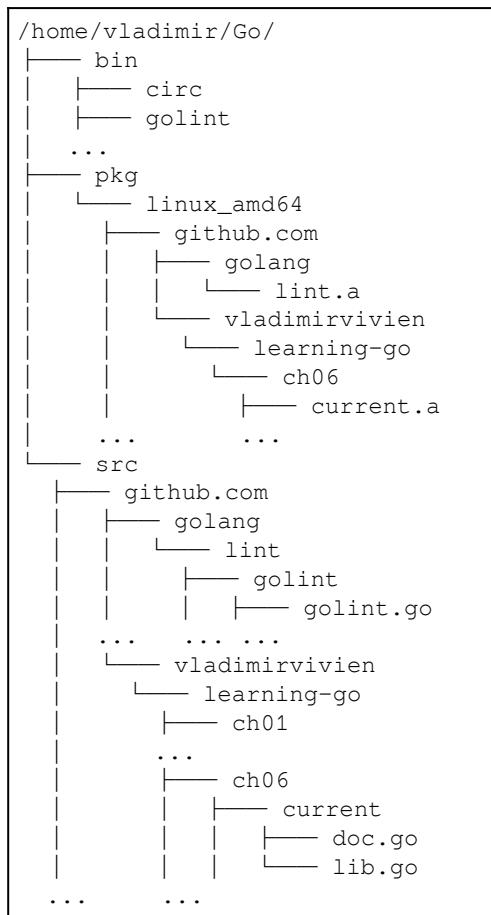
The workspace

Another important concept to understand when discussing packages is that of the *Go workspace*. The workspace is simply an arbitrary directory that serves as a namespace used to resolve packages during certain tasks such as compilation. By convention, Go tools expect three specifically named subdirectories in a workspace directory: `src`, `pkg`, and `bin`. These subdirectories store Go source files along with all built package artifacts respectively.

Establishing a static directory location where Go packages are kept together has the following advantages:

- Simple setup with near-zero configuration
- Fast compilation by reducing code search to a known location
- Tools can easily create source graph of code and package artifacts
- Automatic inference and resolution of transitive dependencies from source
- Project setup can be made portable and easily distributable

The following is a partial (and simplified) tree layout of my Go workspace on my laptop with the three subdirectories, `bin`, `pkg`, and `src`, highlighted:



Sample workspace directory

- `bin`: This is an auto-generated directory that stores compiled Go executable artifacts (also known as programs or commands). When Go tools compile and install executable packages, they are placed in this directory. The previous sample workspace shows two binaries listed `circ` and `golint`. It is a recommended practice to add this directory to your operating system's `PATH` environment variable to make your command available locally.

- `pkg`: This directory is also auto-generated to store built package artifacts. When the Go tools build and install non-executable packages, they are stored as object files (with `.a` suffix) in subdirectories with name patterns based on the targeted operating system and architecture. In the sample workspace, the object files are placed under subdirectory `linux_amd64`, which indicates that the object files in this directory were compiled for the Linux operating system running on a 64-bit architecture.
- `src`: This is a user-created directory where the Go source code files are stored. Each subdirectory under `src` is mapped to a package. `src` is the root directory from which all import paths are resolved. The Go tools search that directory to resolve packages referenced in your code during compilation or other activities that depend on the source path. The sample workspace in the previous figure shows two packages: `github.com/golang/lint/golint/` and `github.com/vladimirvivien/learning-go/ch06/current`.



You may be wondering about the `github.com` prefix in the package path shown in the workspace example. It is worth noting there are no naming requirements for the package directories (see the *Naming packages* section). A package can have any arbitrary name. However, Go recommends certain conventions that help with global namespace resolution and package organization.

Creating a workspace

Creating a workspace is as simple as setting an operating system environment named `GOPATH` and assigning to it the root path of the location of the workspace directory. On a Linux machine, for instance, where the root directory for the workspace is `/home/username/Go`, the workspace would be set as:

```
$> export GOPATH=/home/username/Go
```

When setting up the `GOPATH` environment variable, it is possible to specify multiple locations where packages are stored. Each directory is separated by an OS-dependent path delimiter character (in other words, colon for Linux/Unix, semi-colon for Windows) as shown below:

```
$> export GOPATH=/home/myaccount/Go;/home/myaccount/poc/Go
```

The Go tools will search all listed locations in the `GOPATH` when resolving package names. The Go compiler will, however, only store compiled artifacts, such as object and binary files, in the first directory location assigned to `GOPATH`.



The ability to configure your workspace by simply setting an OS environmental variable has tremendous advantages. It gives developers the ability to dynamically set the workspace at compile time to meet certain workflow requirements. For instance, a developer may want to test an unverified branch of code prior to merging it. He or she may want to set up a temporary workspace to build that code as follows (Linux): \$> `GOPATH=/temporary/go/workspace/path go build`

The import path

Before moving on to the detail of setting up and using packages, one last important concept to cover is the notion of an *import path*. The relative path of each package, under workspace path `$GOPATH/src`, constitutes a global identifier known as the package's `import path`. This implies that no two packages can have the same import path values in a given workspace.

Let us go back to our simplified directory tree from earlier. For instance, if we set the workspace to some arbitrary path value such as `GOPATH=/home/username/Go`:

```
/home/username/Go
  └── foo
      ├── ablt.go
      └── bazz
          ├── quux.go
          └── qux.go
```

From the sample workspace illustrated above, the directory path of the packages is mapped to their respective import paths as shown in the following table:

Directory Path	Import Path
<code>/home/username/Go/foo</code>	<code>"foo"</code>
<code>/home/username/Go/foo/bar</code>	<code>"foo/bar"</code>
<code>/home/username/Go/foo/bar/bazz</code>	<code>"foo/bar/bazz"</code>

Creating packages

Until now, the chapter has covered the rudimentary concepts of the Go package; now it is time to dive deeper and look at the creation of Go code contained in packages. One of the main purposes of a Go package is to abstract out and aggregate common logic into sharable code units. Earlier in the chapter, it was mentioned that a group of Go source files in a directory is considered to be a package. While this is technically true, there is more to the concept of a Go package than just shoving a bunch of files in a directory.

To help illustrate the creation of our first packages, we will enlist the use of example source code found in github.com/vladimirvivien/learning-go/ch06. The code in that directory defines a set of functions to help calculate electrical values using *Ohm's Law*. The following shows the layout of the directories that make up the packages for the example (assuming they are saved in some workspace directory \$GOPATH/src):

```
github.com/vladimirvivien/learning-go/ch06
├── current
│   ├── curr.go
│   └── doc.go
├── power
│   ├── doc.go
│   ├── ir
│   │   └── power.go
│   └── powlib.go
└── vr
    └── power.go
resistor
├── doc.go
└── lib.go
└── res_equivalence.go
└── res.go
└── res_power.go
volt
├── doc.go
└── volt.go
```

Package layout for Ohm's Law example

Each directory, in the previous tree, contains one or more Go source code files that define and implement the functions, and other source code elements, that will be arranged into packages and be made reusable. The following table summarizes the import paths and package information extracted from preceding workspace layout:

Import Path	Package
"github.com/vladimirvivien/learning-go/ch06/ current "	current
"github.com/vladimirvivien/learning-go/ch06/ power "	power
"github.com/vladimirvivien/learning-go/ch06/ power/ir "	ir
"github.com/vladimirvivien/learning-go/ch06/ power/vr "	vr
"github.com/vladimirvivien/learning-go/ch06/ resistor "	resistor
"github.com/vladimirvivien/learning-go/ch06/ volt "	volt

While there are no naming requirements, it is sensible to name package directories to reflect their respective purposes. From the previous table, each package in the example is named to represent an electrical concept, such as current, power, resistor, and volt. The *Naming packages* section will go into further detail about package naming conventions.

Declaring the package

Go source files must declare themselves to be part of a package. This is done using the package clause, as the first legal statement in a Go source file. The declared package consists of the package keyword followed by a name identifier. The following shows source file `volt.go` from the `volt` package:

```
package volt

func V(i, r float64) float64 {
    return i * r
}

func Vser(volts ...float64) (Vtotal float64) {
    for _, v := range volts {
        Vtotal = Vtotal + v
    }
    return
}

func Vpi(p, i float64) float64 {
    return p / i
}
```

```
}
```

golang.fyi/ch06/volt/volt.go

The package identifier in the source file can be set to any arbitrary value. Unlike, say, Java, the name of the package does not reflect the directory structure where the source file is located. While there are no requirements for the package name, it is an accepted convention to name the package identifier the same as the directory where the file is located. In our previous source listing, the package is declared with identifier `volt` because the file is stored inside the `volt` directory.

Multi-File packages

The logical content of a package (source code elements such as types, functions, variables, and constants) can physically scale across multiple Go source files. A package directory can contain one or more Go source files. For instance, in the following example, package `resistor` is unnecessarily split among several Go source files to illustrate this point:

```
package resistor
func recip(val float64) float64 {
    return 1 / val
}
golang.fyi/ch06/resistor/lib.go
```

```
package resistor
func Rser(resists ...float64) (Rtotal float64) {
    for _, r := range resists {
        Rtotal = Rtotal + r
    }
    return
}
func Rpara(resists ...float64) (Rtotal float64) {
    for _, r := range resists {
        Rtotal = Rtotal + recip(r)
    }
    return
}
golang.fyi/ch06/resistor/res_equivalence.go
```

```
package resistor
func R(v, i float64) float64 {
    return v / i
}
golang.fyi/ch06/resistor/res.go
```

```
package resistor
func Rvp(v, p float64) float64 {
    return (v * v) / p
}
golang.fyi/ch06/resistor/res_power.go
```

Each file in the package must have a package declaration with the same name identifier (in this case `resistor`). The Go compiler will stitch all elements from all of the source files together to form one logical unit within a single scope that can be used by other packages.

It is important to point out that compilation will fail if the package declaration is not identical across all source files in a given directory. This is understandable, as the compiler expects all files in a directory to be part of the same package.

Naming packages

As mentioned earlier, Go expects each package in a workspace to have a unique fully qualified import path. Your program may have as many packages as you want and your package structure can be as deep as you like in the workspace. However, idiomatic Go prescribes some **rules** for the naming and organization of your packages to make creating and using packages simple.

Use globally unique namespaces

Firstly, it is a good idea to fully qualify the import path of your packages in a global context, especially if you plan to share your code with others. Consider starting the name of your import path with a namespace scheme that uniquely identifies you or your organization. For instance, company *Acme, Inc.* may choose to start all of their Go package names with `acme.com/apps`. So a fully qualified import path for a package would be `"acme.com/apps/foo/bar"`.



Later in this chapter, we will see how package import paths can be used when integrating Go with source code repository services such as GitHub.

Add context to path

Next, as you devise a naming scheme for your package, use the package's path to add context to the name of your package name. The context in the name should start generic and get more specific from left to right. As an example, let us refer to the import paths for the power package (from the example earlier). The calculation of power values is split among three sub-packages shown as follows:

- `github.com/vladimirvivien/learning-go/ch06/power`
- `github.com/vladimirvivien/learning-go/ch06/power/ir`
- `github.com/vladimirvivien/learning-go/ch06/power/vr`

The parent path `power` contains package members with broader context. The sub-packages `ir` and `vr` contain members that are more specific with narrower contexts. This naming pattern is used heavily in Go, including the built-in packages such as the following:

- `crypto/md5`
- `net/http`
- `net/http/httputil`
- `reflect`

Note a package depth of one is a perfectly legitimate package name (see `reflect`) as long as it captures both context and the essence of what it does. Again, keep things simple.

Avoid the temptation of nesting your packages beyond a depth of more than three inside your namespace. This temptation will be especially strong if you are a Java developer used to long nested package names.

Use short names

When reviewing the names of built-in Go packages, one thing you will notice is the brevity of the names compared to other languages. In Go, a package is considered to be a collection of code that implements a specific set of closely related functionalities. As such, the import paths of your packages should be succinct and reflect what they do without being excessively long. Our example source code exemplifies this by naming the package directory with short names such as `volt`, `power`, `resistance`, `current`. In their respective contexts, each directory name states exactly what the package does.

The short name rule is rigorously applied in the built-in packages of Go. For instance, following are several package names from Go's built-in packages: `log`, `http`, `xml`, and `zip`. Each name readily identifies the purpose of the package.



Short package names have the advantage of reducing keystrokes in larger code bases. However, having short and generic package names also has the disadvantage of being prone to import path clashes where developers in a large project (or developers of open source libraries) may end up using the same popular names (in other words, `log`, `util`, `db`, and so on) in their code. As we will see later in the chapter, this can be handled using named import paths.

Building packages

The Go tools reduce the complexity of compiling your code by applying certain conventions and sensible defaults. Although a full discussion of Go's build tool is beyond the scope of this section (or chapter), it is useful to understand the purpose and use of the `build` and `install` tools. In general, the use of the build and install tools is as follows:

```
$> go build [<package import path>]
```

The `import` path can be explicitly provided or omitted altogether. The `build` tool accepts the `import` path expressed as either fully qualified or relative paths. Given a properly setup workspace, the following are all equivalent ways to compile package `volt`, from the earlier example:

```
$> cd $GOPATH/src/github.com/vladimirvivien/learning-go
$> go build ./ch06/volt
$> cd $GOPATH/src/github.com/vladimirvivien/learning-go/ch06
$> go build ./volt
$> cd $GOPATH/src/github.com/vladimirvivien/learning-go/ch06/volt
$> go build .
$> cd $GOPATH/src/
$> go build github.com/vladimirvivien/learning-go/ch06/current /volt
```

The `go build` command above will compile all Go source files and their dependencies found in directory `volt`. Furthermore, it is also possible to build all of your packages and sub-packages in a given directory using the wildcard parameter appended to an import path shown as follows:

```
$> cd $GOPATH/src/github.com/vladimirvivien/learning-go/ch06
$> go build ./...
```

The previous will build all packages and sub-packages found in the directory `$GOPATH/src/github.com/vladimirvivien/learning-go/ch06`.

Installing a package

By default, the build command outputs its results into a tool-generated temporary directory that is lost after the build process completes. To actually generate a usable artifact, you must use the `install` tool to keep a copy of the compiled object files.

The `install` tool has the exact semantics as the build tool:

```
$> cd $GOPATH/src/github.com/vladimirvivien/learning-go/ch06  
$> go install ./volt
```

In addition to compiling the code, it also saves and outputs the result to workspace location `$GOPATH/pkg` as shown in the following:

```
$GOPATH/pkg/linux_amd64/github.com/vladimirvivien/learning-go/  
└── ch06  
    └── volt.a
```

The generated object files (with the `.a` extension) allow the package to be reused and linked against other packages in the workspace. Later in the chapter, we will examine how to compile executable programs.

Package visibility

Regardless of the number of source files declared to be part of a package, all source code elements (types, variables, constants, and functions), declared at a package level, share a common scope. Therefore, the compiler will not allow an element identifier to be re-declared more than once in the entire package. Let us use the following code snippets to illustrate this point, assuming both source files are part of the same package

`$GOPATH/src/foo:`

package foo var (bar int = 12) func qux () { bar += bar } foo/file1.go	package foo var bar struct{ x, y int } func quux() { bar = bar * bar } foo/file2.go
---	--

Illegal variable identifier re-declaration

Although they are in two separate files, the declaration of variables with identifier `bar` is illegal in Go. Since the files are part of the same package, both identifiers have the same scope and therefore clash.

The same is true for function identifiers. Go does not support the overloading of function names within the same scope. Therefore, it is illegal to have a function identifier used more than once, regardless of the function's signature. If we assume the following code appears in two different source files within the same package, the following snippet would be illegal:

package foo var (bar int = 12) func qux () { bar += bar } foo/file1.go	package foo var (fooVal int = 12) func qux (inc int) int { return fooVal += inc } foo/file1.go
---	---

Illegal function identifier re-declaration

In the previous code snippets, function name identifier `qux` is used twice. The compiler will fail the compilation even though both functions have different signatures. The only way to fix this is to change the name.

Package member visibility

The usefulness of a package is its ability to expose its source elements to other packages. Controlling the visibility of elements of a package is simple and follows this rule: *capitalized identifiers are exported automatically*. This means any type, variable, constant, or function with capitalized identifiers is automatically visible from outside of the package where it is declared.

Referring to the Ohm's Law example, described earlier, the following illustrates this functionality from the package `resistor` (found in github.com/vladimirvivien/learning-go/ch06/resistor):

Code	Description
package resistor func R(v, i float64) float64 { return v / i }	Function <code>R</code> is automatically exported and can be accessed from other packages as: <code>resistor.R()</code>

```
package resistor
func recip(val
float64) float64 {
    return 1 / val
}
```

Function identifier `recip` is in all lowercase and therefore is not exported. Though accessible within its own scope, the function will not be visible from within other packages.

It is worth restating that members within the same package are always visible to each other. In Go, there are no complicated visibility structures of private, friend, default, and so on, as is found in other languages. This frees the developer to concentrate on the solution being implemented rather than modeling visibility hierarchies.

Importing package

At this point, you should have a good understanding of what a package is, what it does, and how to create one. Now, let us see how to use a package to import and reuse its members. As you will find in several other languages, the keyword `import` is used to import source code elements from an external package. It allows the importing source to access exported elements found in the imported package (see the *Package scope and visibility* section earlier in the chapter). The general format for the import clause is as follows:

```
import [package name identifier] "<import path>"
```

Notice that the import path must be enclosed within double quotes. The `import` statement also supports an optional package identifier that can be used to explicitly name the imported package (discussed later). The import statement can also be written as an import block, as shown in the following format. This is useful where there are two or more import packages listed:

```
import (
    [package name identifier] "<import path>"
)
```

The following source code snippet shows the import declaration block in the Ohm's Law examples introduced earlier:

```
import (
    "flag"
    "fmt"
    "os"

    "github.com/vladimirvivien/learning-go/ch06/current"
    "github.com/vladimirvivien/learning-go/ch06/power"
    "github.com/vladimirvivien/learning-go/ch06/power/ir"
    "github.com/vladimirvivien/learning-go/ch06/power/vr"
    "github.com/vladimirvivien/learning-go/ch06/volt"
)

golang.fyi/ch06/main.go
```

Often the name identifiers of imported packages are omitted, as done above. Go then applies the name of the last directory of the import path as the name identifier for the imported package, as shown, for some packages, in the following table:

Import Path	Package name
flag	flag
github.com/vladimirvivien/learning-go/ch06/current	current
github.com/vladimirvivien/learning-go/ch06/power/ir	ir
github.com/vladimirvivien/learning-go/ch06/volt	volt

The dot notation is used to access exported members of an imported package. In the following source code snippet, for instance, method `volt.V()` is invoked from imported package `"github.com/vladimirvivien/learning-go/ch06/volt"`:

```
...
import "github.com/vladimirvivien/learning-go/ch06/volt"
func main() {
    ...
    switch op {
    case "V", "v":
        val := volt.V(i, r)
    ...
}
```

golang.fyi/ch06/main.go

Specifying package identifiers

As was mentioned, an `import` declaration may explicitly declare a name identifier for the import, as shown in the following import snippet:

```
import res "github.com/vladimirvivien/learning-go/ch06/resistor"
```

Following the format described earlier, the name identifier is placed before the import path as shown in the preceding snippet. A named package can be used as a way to shorten or customize the name of a package. For instance, in a large source file with numerous usage of a certain package, this can be a welcome feature to reduce keystrokes.

Assigning a name to a package is also a way to avoid package identifier collisions in a given source file. It is conceivable to import two or more packages, with different import paths, that resolve to the same package names. As an example, you may need to log information with two different logging systems from different libraries, as illustrated in the following code snippet:

```
package foo
import (
    flog "github.com/woom/bat/logger"
    hlog "foo/bar/util/logger"
)

func main() {
    flog.Info("Programm started")
    err := doSomething()
    if err != nil {
        hlog.SubmitError("Error - unable to do something")
    }
}
```

As depicted in the previous snippet, both logging packages will resolve to the same name identifier of `"logger"` by default. To resolve this, at least one of the imported packages must be assigned a name identifier to resolve the name clash. In the previous example, both import paths were named with a meaningful name to help with code comprehension.

The dot identifier

A package can optionally be assigned a dot (period) as its identifier. When an import statement uses the dot identifier (.) for an import path, it causes members of the imported package to be merged in scope with that of the importing package. Therefore, imported members may be referenced without additional qualifiers. So if package logger is imported with the dot identifier in the following source code snippet, when accessing exported member function SubmitError from the logger package, the package name is omitted:

```
package foo

import (
    . "foo/bar/util/logger"
)

func main() {
    err := doSomething()
    if err != nil {
        SubmitError("Error - unable to do something")
    }
}
```

While this feature can help reduce repetitive keystrokes, it is not an encouraged practice. By merging the scope of your packages, it becomes more likely to run into identifier collisions.

The blank identifier

When a package is imported, it is a requirement that one of its members be referenced in the importing code at least once. Failure to do so will result in a compilation error. While this feature helps simplify package dependency resolution, it can be cumbersome, especially in the early phase of a developing code.

Using the blank identifier (similar to variable declarations) causes the compiler to bypass this requirement. For instance, the following snippet imports the built-in package fmt; however, it never uses it in the subsequent source code:

```
package foo
import (
    _ "fmt"
    "foo/bar/util/logger"
)

func main() {
```

```
    err := doSomething()
    if err != nil {
        logger.Submit("Error - unable to do something")
    }
}
```

A common idiom for the blank identifier is to load packages for their side effects. This relies on the initialization sequence of packages when they are imported (see the following *Package initialization* section). Using the blank identifier will cause an imported package to be initialized even when none of its members can be referenced. This is used in contexts where the code is needed to silently run certain initialization sequences.

Package initialization

When a package is imported, it goes through a series of initialization sequences before its members are ready to be used. Package-level variables are initialized using dependency analysis that relies on lexical scope resolution, meaning variables are initialized based on their declaration order and their resolved transitive references to each other. For instance, in the following snippet, the resolved variable declaration order in package `foo` will be `a`, `y`, `b`, and `x`:

```
package foo
var x = a + b(a)
var a = 2
var b = func(i int) int {return y * i}
var y = 3
```

Go also makes use of a special function named `init` that takes no arguments and returns no result values. It is used to encapsulate custom initialization logic that is invoked when the package is imported. For instance, the following source code shows an `init` function used in the `resistor` package to initialize function variable `Rpi`:

```
package resistor

var Rpi func(float64, float64) float64

func init() {
    Rpi = func(p, i float64) float64 {
        return p / (i * i)
    }
}

func Rvp(v, p float64) float64 {
    return (v * v) / p
```

```
}
```

```
golang.fyi/ch06/resistor/res_power.go
```

In the preceding code, the `init` function is invoked after the package-level variables are initialized. Therefore, the code in the `init` function can safely rely on the declared variable values to be in a stable state. The `init` function is special in the following ways:

- A package can have more than one `init` functions defined
- You cannot directly access declared `init` functions at runtime
- They are executed in the lexical order they appear within each source file
- The `init` function is a great way to inject logic into a package that gets executed prior to any other functions or methods.

Creating programs

So far in the book, you have learned how to create and bundle Go code as reusable packages. A package, however, cannot be executed as a standalone program. To create a program (also known as a command), you take a package and define an entry point of execution as follows:

- Declare (at least one) source file to be part of a special package called `main`
- Declare one function name `main()` to be used as the entry point of the program

The function `main` takes no argument nor returns any value. The following shows the abbreviated source code for the `main` package used in the Ohm's Law example (from earlier). It uses the package `flag`, from Go's standard library, to parse program arguments formatted as `flag`:

```
package main
import (
    "flag"
    "fmt"
    "os"

    "github.com/vladimirvivien/learning-go/ch06/current"
    "github.com/vladimirvivien/learning-go/ch06/power"
    "github.com/vladimirvivien/learning-go/ch06/power/ir"
    "github.com/vladimirvivien/learning-go/ch06/power/vr"
    res "github.com/vladimirvivien/learning-go/ch06/resistor"
    "github.com/vladimirvivien/learning-go/ch06/volt"
)
```

```
var (
    op string
    v float64
    r float64
    i float64
    p float64

    usage = "Usage: ./circ <command> [arguments]\n" +
        "Valid command { V | Vpi | R | Rvp | I | Ivp | "+
        "P | Pir | Pvr }"
)

func init() {
    flag.Float64Var(&v, "v", 0.0, "Voltage value (volt)")
    flag.Float64Var(&r, "r", 0.0, "Resistance value (ohms)")
    flag.Float64Var(&i, "i", 0.0, "Current value (amp)")
    flag.Float64Var(&p, "p", 0.0, "Electrical power (watt)")
    flag.StringVar(&op, "op", "V", "Command - one of { V | Vpi | "+
        "R | Rvp | I | Ivp | P | Pir | Pvr }")
}

func main() {
    flag.Parse()
    // execute operation
    switch op {
    case "V", "v":
        val := volt.V(i, r)
        fmt.Printf("V = %0.2f * %0.2f = %0.2f volts\n", i, r, val)
    case "Vpi", "vpi":
        val := volt.Vpi(p, i)
        fmt.Printf("Vpi = %0.2f / %0.2f = %0.2f volts\n", p, i, val)
    case "R", "r":
        val := res.R(v, i)
        fmt.Printf("R = %0.2f / %0.2f = %0.2f Ohms\n", v, i, val)
    case "I", "i":
        val := current.I(v, r))
        fmt.Printf("I = %0.2f / %0.2f = %0.2f amps\n", v, r, val)
    ...
    default:
        fmt.Println(usage)
        os.Exit(1)
    }
}
```

golang.fyi/ch06/main.go

The previous listing shows the source code of the `main` package and the implementation of the function `main` which gets executed when the program runs. The Ohm's Law program accepts command-line arguments that specify which electrical operation to execute (see the following *Accessing program arguments* section). The function `init` is used to initialize parsing of the program flag values. The function `main` is set up as a big switch statement block to select the proper operation to execute based on the selected flags.

Accessing program arguments

When a program is executed, the Go runtime makes all command-line arguments available as a slice via package variable `os.Args`. For instance, when the following program is executed, it prints all command-line arguments passed to the program:

```
package main
import (
    "fmt"
    "os"
)

func main() {
    for _, arg := range os.Args {
        fmt.Println(arg)
    }
}
```

golang.fyi/ch06-args/hello.go

The following is the output of the program when it is invoked with the shown arguments:

```
$> go run hello.go hello world how are you?
/var/folders/.../exe/hello
hello
world
how
are
you?
```

Note that the command-line argument "hello world how are you?", placed after the program's name, is split as a space-delimited string. Position 0 in slice `os.Args` holds the fully qualified name of the program's binary path. The rest of the slice stores each item in the string respectively.

The `flag` package, from Go's standard library, uses this mechanism internally to provide processing of structured command-line arguments known as flags. In the Ohm's Law example listed earlier, the `flag` package is used to parse several flags, as listed in the following source snippet (extracted from the full listing earlier):

```
var (
    op string
    v float64
    r float64
    i float64
    p float64
)

func init() {
    flag.Float64Var(&v, "v", 0.0, "Voltage value (volt)")
    flag.Float64Var(&r, "r", 0.0, "Resistance value (ohms)")
    flag.Float64Var(&i, "i", 0.0, "Current value (amp)")
    flag.Float64Var(&p, "p", 0.0, "Electrical power (watt)")
    flag.StringVar(&op, "op", "V", "Command - one of { V | Vpi | "+
        " R | Rvp | I | Ivp | P | Pir | Pvr }")
}
func main(){
    flag.Parse()
    ...
}
```

The snippet shows function `init` used to parse and initialize expected flags "`v`", "`i`", "`p`", and "`op`" (at runtime, each flag is prefixed with a minus sign). The initialization functions in package `flag` sets up the expected type, the default value, a flag description, and where to store the parsed value for the flag. The `flag` package also supports the special flag "`help`", used to provide helpful hints about each flag.

`flag.Parse()`, in the function `main`, is used to start the process of parsing any flags provided as command-line. For instance, to calculate the current of a circuit with 12 volts and 300 ohms, the program takes three flags and produces the shown output:

```
$> go run main.go -op I -v 12 -r 300
I = 12.00 / 300.00 = 0.04 amps
```

Building and installing programs

Building and installing Go programs follow the exact same procedures as building a regular package (as was discussed earlier in the *Building and installing packages* section). When you build source files of an executable Go program, the compiler will generate an executable binary file by transitively linking all the dependencies declared in the `main` package. The build tool will name the output binary, by default the same name as the directory where the Go program source files are located.

For instance, in the Ohm's Law example, the file `main.go`, which is located in the directory `github.com/vladimirvivien/learning-go/ch06`, is declared to be part of the `main` package. The program can be built as shown in the following:

```
$> cd $GOPATH/src/github.com/vladimirvivien/learning-go/ch06  
$> go build .
```

When the `main.go` source file is built, the build tool will generate a binary named `ch06` because the source code for the program is located in a directory with that name. You can control the name of the binary using the output flag `-o`. In the following example, the build tool creates a binary file named `ohms`.

```
$> cd $GOPATH/src/github.com/vladimirvivien/learning-go/ch06  
$> go build -o ohms
```

Lastly, installing a Go program is done in exactly the same way as installing a regular package using the Go `install` command:

```
$> cd $GOPATH/src/github.com/vladimirvivien/learning-go/ch06  
$> go install .
```

When a program is installed using the Go `install` command, it will be built, if necessary, and its generated binary will be saved in the `$GOPATH/bin` directory. Adding the workspace `bin` directory to your OS's `$PATH` environment variable will make your Go program available for execution.



Go-generated programs are statically linked binaries. They require no additional dependencies to be satisfied to run. However, Go-compiled binaries include the Go runtime. This is the set of operations that handle functionalities such as garbage collection, type information, reflection, goroutines scheduling, and panic management. While a comparable C program would be orders of magnitudes smaller, Go's runtime comes with the tools that make Go enjoyable.

Remote packages

One of the tools that is shipped with Go allows programmers to retrieve packages directly from remote source code repositories. Go, by default, readily supports integration with version control systems including the following:

- Git (`git, http://git-scm.com/`)
- Mercurial (`hg, http://mercurial.selenic.com/`)
- Subversion (`svn, http://subversion.apache.org/`)
- Bazaar (`bzr, http://bazaar.canonical.com/`)



In order for Go to pull package source code from a remote repository, you must have a client for that version control system installed as a command on your operating system's execution path. Under the cover, Go launches the client to interact with the source code repository server.

The `get` command-line tool allows programmers to retrieve remote packages using a fully qualified project path as the import path for the package. Once the package is downloaded, it can be imported for use in local source files. For instance, if you wanted to include one of the packages from the Ohm's Law example from preceding snippet, you would issue the following command from the command-line:

```
$> go get github.com/vladimirvivien/learning-go/ch06/volt
```

The `go get` tool will download the specified import path along with all referenced dependencies. The tool will then build and install the package artifacts in `$GOPATH/pkg`. If the import path happens to be a program, `go get` will generate the binary in `$GOPATH/bin` as well as any referenced packages in `$GOPATH/pkg`.

Summary

This chapter presented an extensive look into the notion of source code organization and packages. Readers learned about the Go workspace and the import path. Readers were also introduced to the creation of packages and how to import packages to achieve code reusability. The chapter introduced mechanisms such as visibility of imported members and package initialization. The last portion of the chapter discussed the steps that are necessary to create an executable Go program from packaged code.

This was a lengthy chapter, and deservedly so to do justice to such a broad topic as package creation and management in Go. The next chapter returns to the Go types discussion with a detailed treatment of the composite types, such as array, slice, struct, and map.

7

Composite Types

In prior chapters, you may have caught glimpses of the use of composite types such as arrays, slices, maps, and structs in some of the sample code. While early exposure to these types may have left you curious, rest assured in this chapter you will get a chance to learn all about these composite types. This chapter continues what started in Chapter 4, *Data Types*, with discussions covering the following topics:

- The array type
- The slice type
- The map type
- The struct type

The array type

As you would find in other languages, Go arrays are containers for storing sequenced values of the same type that are numerically indexed. The following code snippet shows samples of variables that are assigned array types:

```
var val [100]int
var days [7]string
var truth [256]bool
var histogram [5]map[string]int
```

golang.fyi/ch07/arrtypes.go

Notice the types that are assigned to each variable in the previous example are specified using the following type format:

[<length>]<element_type>

The type definition of an array is composed of its length, enclosed within brackets, followed by the type of its stored elements. For instance, the `days` variable is assigned a type `[7]string`. This is an important distinction as Go's type system considers two arrays, storing the same type of elements but with different lengths, to be of different types. The following code illustrates this situation:

```
var days [7]string
var weekdays [5]string
```

Even though both variables are arrays with elements of type `string`, the type system considers the `days` and `weekdays` variables as different types.



Later in the chapter, you will see how this type restriction is mitigated with the use of the slice type instead of arrays.

Array types can be defined to be multi-dimensions. This is done by combining and nesting the definition of one-dimensional array types as shown in the following snippet:

```
var board [4][2]int
var matrix [2][2][2][2] byte
```

golang.fyi/ch07/arrtypes.go

Go does not have a separate type for multi-dimensional arrays. An array with more than one dimension is composed of one-dimensional arrays that are nested within each other. The next section covers how single and multi-dimensional arrays are initialized.

Array initialization

When an array variable is not explicitly initialized, all of its elements will be assigned the zero-value for the declared type of the elements. An array can be initialized with a composite literal value with the following general format:

<array_type>{<comma-separated list of element values>}

The literal value for an array is composed of the array type definition (discussed in the previous section) followed by a set of comma-separated values, enclosed in curly brackets, as illustrated by the following code snippet, which shows several arrays being declared and initialized:

```
var val [100]int = [100]int{44,72,12,55,64,1,4,90,13,54}
var days [7]string = [7]string{
    "Monday",
    "Tuesday",
    "Wednesday",
    "Thursday",
    "Friday",
    "Saturday",
    "Sunday",
}
var truth = [256]bool{true}
var histogram = [5]map[string]int {
    map[string]int{"A":12, "B":1, "D":15},
    map[string]int{"man":1344, "women":844, "children":577, ...},
}
```

golang.fyi/ch07/arrinit.go

The number of elements in the literal must be less than or equal to the size declared in the array type. If the array defined is multi-dimensional, it can be initialized using literal values by nesting each dimension within the enclosing brackets of another, as shown in the following example snippets:

```
var board = [4][2]int{
    {33, 23},
    {62, 2},
    {23, 4},
    {51, 88},
}
var matrix = [2][2][2][2]byte{
    {{4, 4}, {3, 5}, {{55, 12}, {22, 4}}},
    {{2, 2}, {7, 9}, {{43, 0}, {88, 7}}},
}
```

golang.fyi/ch07/arrinit.go

The following snippet shows two additional ways that array literals can be specified. The length of an array may be omitted and replaced by ellipses during initialization. The following will assign type [5]string to variable `weekdays`:

```
var weekdays = [...]string{
    "Monday",
    "Tuesday",
    "Wednesday",
    "Thursday",
    "Friday",
}
```

The literal value of an array can also be indexed. This is useful if you want to initialize only certain array elements while allowing others to be initialized with their natural zero-value. The following specifies the initial values for elements at positions 0, 2, 4, 6, 8. The remaining elements will be assigned the empty string:

```
var msg = [12]rune{0: 'H', 2: 'E', 4: 'L', 6: 'O', 8: '!'}
```

Declaring named array types

The type of an array can become awkward for reuse. For each declaration, it becomes necessary to repeat the declaration, which can be error prone. The way to handle this idiomatically is to alias array types using type declarations. To illustrate how this works, the following code snippet declares a new named type, `matrix`, using a multi-dimension array as its underlying type:

```
type matrix [2][2][2][2]byte

func main() {
    var mat1 matrix
    mat1 = initMat()
    fmt.Println(mat1)
}

func initMat() matrix {
    return matrix{
        {{{4, 4}, {3, 5}}, {{55, 12}, {22, 4}}},
        {{{2, 2}, {7, 9}}, {{43, 0}, {88, 7}}},
    }
}
```

golang.fyi/ch07/arrtype_dec.go

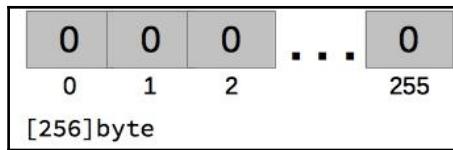
The declared named type, `matrix`, can be used in all contexts where its underlying array type is used. This allows a simplified syntax that promotes reuse of the complex array type.

Using arrays

Arrays are static entities that cannot grow or shrink in size once they are declared with a specified length. Arrays are a great option when a program needs to allocate a block of sequential memory of a predefined size. When a variable of an array type is declared, it is ready to be used without any further allocation semantics.

So the following declaration of the `image` variable would allocate a memory block composed of 256 adjacent `int` values initialized with zeroes, as shown in the following figure:

```
var image [256]byte
```



Similar to C and Java, Go uses the square brackets index expression to access values stored in an array variable. This is done by specifying the variable identifier followed by an index of the element enclosed within the square brackets, as shown in the following code sample:

```
p := [5]int{122, 6, 23, 44, 6}
p[4] = 82
fmt.Println(p[0])
```

The previous code updates the fifth element and prints the first element in the array.

Array length and capacity

The built-in `len` function returns the declared length of an array type. The built-in `cap` function can be used on an array to return its capacity. For instance, in the following source snippet, the array `seven` of type `[7]string` will return 7 as its length and capacity:

```
func main() {
    seven := [7]string{"grumpy", "sleepy", "bashful"}
    fmt.Println(len(seven), cap(seven))
}
```

For arrays, the `cap()` function always returns the same value as `len()`. This is because the maximum capacity of an array value is its declared length. The capacity function is better suited for use with the slice type (discussed later in the chapter).

Array traversal

Array traversal can be done using the traditional `for` statement or with the more idiomatic `for...range` statement. The following snippet of code shows array traversal done with both the `for` statement, to initialize an array with random numbers in `init()`, and the `for range` statement used to realize the `max()` function:

```
const size = 1000
var nums [size]int

func init() {
    rand.Seed(time.Now().UnixNano())
    for i := 0; i < size; i++ {
        nums[i] = rand.Intn(10000)
    }
}

func max(nums [size]int) int {
    temp := nums[0]
    for _, val := range nums {
        if val > temp {
            temp = val
        }
    }
    return temp
}
```

golang.fyi/ch07/arrmax_iter.go

In the traditional `for` statement, the loop's index variable `i` is used to access the value of the array using the index expression `num[i]`. In the `for...range` statement, in the `max` function, the iterated value is stored in the `val` variable with each pass of the loop and the index is ignored (assigned to the blank identifier). If you do not understand how `for` statements work, refer to Chapter 3, *Go Control Flow*, for a thorough explanation of the mechanics of loops in Go.

Array as parameters

Arrays values are treated as a single unit. An array variable is not a pointer to a location in memory, but rather represents the entire block of memory containing the array elements. This has the implications of creating a new copy of an array value when the array variable is reassigned or passed in as a function parameter.

This could have unwanted side effects on memory consumption for a program. One fix for is to use pointer types to reference array values. In the following example, a named type, `numbers`, is declared to represent array type `[1024 * 1024] int`. Instead of taking the array value directly as parameters, functions `initialize()` and `max()` receive a pointer of type `*numbers`, as shown in the following source snippet:

```
type numbers [1024 * 1024]int
func initialize(nums *numbers) {
    rand.Seed(time.Now().UnixNano())
    for i := 0; i < size; i++ {
        nums[i] = rand.Intn(10000)
    }
}
func max(nums *numbers) int {
    temp := nums[0]
    for _, val := range nums {
        if val > temp {
            temp = val
        }
    }
    return temp
}
func main() {
    var nums *numbers = new(numbers)
    initialize(nums)
}
```

golang.fyi/ch07/arrptr.go

The previous code uses the built-in function `new(numbers)` to initialize the array elements with their zero values and obtain a pointer to that array as shown in `main()`. So when the functions `initialize` and `max` are invoked, they will receive the address (a copy of it) of the array instead of the entire 100K-sized array.

Before changing the subject, it should be noted that a composite literal array value can be initialized with the address operator `&` to initialize and return a pointer for the array, as shown in the following example. In the snippet, composite literal `&galaxies{ . . . }` returns pointer `*galaxies`, initialized with the specified element values:

```
type galaxies [14]string
func main() {
    namedGalaxies = &galaxies{
        "Andromeda",
        "Black Eye",
        "Bode's",
        ...
    }
    printGalaxies(namedGalaxies)
}
```

golang.fyi/ch07/arraddr.go

The array type is a low-level storage construct in Go. Arrays, for instance, are usually used as the basis for storage primitives, where there are strict memory allocation requirements to minimize space consumption. In more common cases however, the slice, covered in the next section, is often used as the more idiomatic way of working with sequenced indexed collections.

The slice type

The slice type is commonly used as the idiomatic construct for indexed data in Go. The slice is more flexible and has many more interesting characteristics than arrays. The slice itself is a composite type with semantics similar to arrays. In fact, a slice uses an array as its underlying data storage mechanism. The general form of a slice type is given as follows:

`[]<element_type>`

The one obvious difference between a slice and an array type is omission of the size in the type declaration, as shown in the following examples:

```
var (
    image []byte
```

```
ids []string
vector []float64
months []string
q1 []string
histogram []map[string]int // slice of map (see map later)
)
```

golang.fyi/ch07/slicetypes.go

The missing size attribute in the slice type indicates the following:

- Unlike arrays, the size of a slice is not fixed
- A slice type represents all sets of the specified element type

This means a slice can theoretically grow unbounded (though in practice this is not true as the slice is backed by an underlying bounded array). A slice of a given element type is considered to be the same type regardless of its underlying size. This removes the restriction found in arrays where the size determines the type.

For instance, the following variables, `months` and `q1`, have the same type of `[]string` and will compile with no problem:

```
var (
    months []string
    q1 []string
)
func print(strs []string) { ... }
func main() {
    print(months)
    print(q1)
}
```

golang.fyi/ch07/slicetypes.go

Similar to arrays, slice types may be nested to create multi-dimensional slices, as shown in the following code snippet. Each dimension can independently have its own size and must be initialized individually:

```
var(
    board [][]int
    graph [][][][][int
)
```

Slice initialization

A slice is represented by the type system as a value (the next section explores the internal representation of a slice). However, unlike the array type, an uninitialized slice has a zero value of *nil*, which means any attempt to access elements of an uninitialized slice will cause a program to panic.

One of the simplest ways to initialize a slice is with a composite literal value using the following format (similar to an array):

<slice_type>{<comma-separated list of element values>}

The literal value for a slice is composed of the slice type followed by a set of comma-separated values, enclosed in curly brackets, that are assigned to the elements of the slice. The following code snippet illustrates several slice variables initialized with composite literal values:

```
var (
    ids []string = []string{"fe225", "ac144", "3b12c"}
    vector = []float64{12.4, 44, 126, 2, 11.5}
    months = []string {
        "Jan", "Feb", "Mar", "Apr",
        "May", "Jun", "Jul", "Aug",
        "Sep", "Oct", "Nov", "Dec",
    }
    // slice of map type (maps are covered later)
    tables = []map[string][]int {
        {
            "age":{53, 13, 5, 55, 45, 62, 34, 7},
            "pay":{124, 66, 777, 531, 933, 231},
        },
    }
    graph = [][][]int{
        {{44}, {3, 5}}, {{55, 12, 3}, {22, 4}},
        {{22, 12, 9, 19}, {7, 9}}, {{43, 0, 44, 12}, {7}}},
    }
)
```

golang.fyi/ch07/sliceinit.go

As mentioned, the composite literal value of a slice is expressed using a similar form as the array. However, the number of elements provided in the literal is not bounded by a fixed size. This implies that the literal can be as large as needed. Under the cover though, Go creates and manages an array of appropriate size to store the values expressed in the literal.

Slice representation

Earlier it was mentioned that the slice value uses an underlying array to store data. The name *slice*, in fact, is a reference to a slice of data segment from the array. Internally, a slice is represented by a composite value with the followings three attributes:

Attribute	Description
a <i>pointer</i>	The pointer is the address of the first element of the slice stored in an underlying array. When the slice value is uninitialized, its pointer value is nil, indicating that it is not pointing to an array yet. Go uses the pointer as the zero value of the slice itself. An uninitialized slice will return nil as its zero value. However, the slice value is not treated as a reference value by the type system. This means certain functions can be applied to a nil slice while others will cause a panic. Once a slice is created, the pointer does not change. To point to a different starting point, a new slice must be created.
a <i>length</i>	The length indicates the number of contiguous elements that can be accessed starting with the first element. It is a dynamic value that can grow up to the capacity of the slice (see capacity next). The length of a slice is always less than or equal to its capacity. Attempts to access elements beyond the length of a slice, without resizing, will result in a panic. This is true even when the capacity is larger than the length.
a <i>capacity</i>	The capacity of a slice is the maximum number of elements that may be stored in the slice, starting from its first element. The capacity of a slice is bounded by the length of the underlying array.

So, when the following variable `halfyr` is initialized as shown:

```
halfyr := []string{"Jan", "Feb", "Mar", "Apr", "May", "Jun"}
```

It will be stored in an array of type `[6]string` with a pointer to the first element, a length, and a capacity of 6, as represented graphically in the following figure:



Slicing

Another way to create a slice value is by slicing an existing array or another slice value (or pointers to these values). Go provides an indexing format that makes it easy to express the slicing operation, as follows:

`<slice or array value>[<low_index>:<high_index>]`

The slicing expression uses the `[:]` operator to specify the low and high bound indices, separated by a colon, for the slice segment.

- The *low* value is the zero-based index where the slice segment starts
- The *high* value is the n^{th} element offset where the segment stops

The following table shows examples of slice expressions by re-slicing the following value:

```
halfyr := []string{"Jan", "Feb", "Mar", "Apr", "May", "Jun"}.
```

Expression	Description
<code>all := halfyr[:]</code>	Omitting the low and high indices in the expression is equivalent to the following: <code>all := halfyr[0 : 6]</code> This produces a new slice segment equal to the original, which starts at index position 0 and stops at offset position 6: <code>["Jan", "Feb", "Mar", "Apr", "May", "Jun"]</code>
<code>q1 := halfyr[:3]</code>	Here the slice expression omits low index value and specifies a slice segment length of 3. It returns new slice, <code>["Jan", "Feb", "Mar"]</code> .
<code>q2 := halfyr[3:]</code>	This creates a new slice segment with the last three elements by specifying the starting index position of 3 and omitting the high bound index value, which defaults to 6.
<code>mapr := halfyr[2:4]</code>	To clear any confusion about slicing expressions, this example shows how to create a new slice with the months "Mar" and "Apr". This returns a slice with the value <code>["Mar", "Apr"]</code> .

Slicing a slice

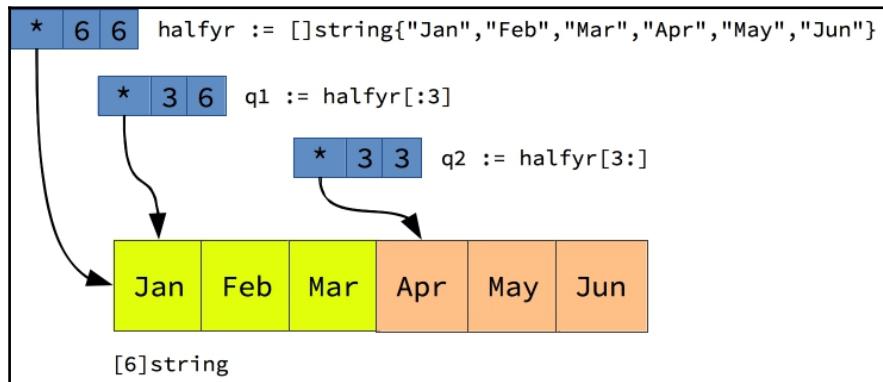
Slicing an existing slice or array value does not create a new underlying array. The new slice creates new pointer location to the underlying array. For instance, the following code shows the slicing of the slice value `halfyr` into two additional slices:

```
var (
    halfyr = []string{
        "Jan", "Feb", "Mar",
        "Apr", "May", "Jun",
    }

    q1 = halfyr[:3]
    q2 = halfyr[3:]
)
```

golang.fyi/ch07/slice_reslice.go

The backing array may have many slices projecting a particular view of its data. The following figure illustrates how slicing in the previous code may be represented visually:



Notice that both slices `q1` and `q2` are pointing to different elements in the same underlying array. Slice `q1` has an initial length of 3 with a capacity of 6. This implies `q1` can be resized up to 6 elements in total. Slice `q2`, however, has a size of 3 and a capacity of 3 and cannot grow beyond its initial size (slice resizing is covered later).

Slicing an array

As mentioned, an array can also be sliced directly. When that is the case, the provided array value becomes the underlying array. The capacity and the length the slices will be calculated using the provided array. The following source snippet shows the slicing of an existing array value called months:

```
var (
    months [12]string = [12]string{
        "Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec",
    }

    halfyrs = months[:6]
    q1 = halfyrs[:3]
    q2 = halfyrs[3:6]
    q3 = months[6:9]
    q4 = months[9:]
)
```

golang.fyi/ch07/slice_reslice_arr.go

Slice expressions with capacity

Lastly, Go's slice expression supports a longer form where the maximum capacity of the slice is included in the expression, as shown here:

<slice_or_array_value>[<low_index>:<high_index>:<max>]

The *max* attribute specifies the index value to be used as the maximum capacity of the new slice. That value may be less than, or equal to, the actual capacity of the underlying array. The following example slices an array with the max value included:

```
var (
    months [12]string = [12]string{
        "Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec",
    }
    summer1 = months[6:9:9]
)
```

golang.fyi/ch07/slice_reslice_arr.go

The previous code snippet creates a new slice value `summer1` with size 3 (starting at index position 6 to 9). The max index is set to position 9, which means the slice has a capacity of 3. If the max was not specified, the maximum capacity would automatically be set to the last position of the underlying array as before.

Making a slice

A slice can be initialized at runtime using the built-in function `make`. This function creates a new slice value and initializes its elements with the zero value of the element type. An uninitialized slice has a nil zero value an indication that it is not pointing an underlying array. Without an explicitly initialization, with a composite literal value or using the `make()` function, attempts to access elements of a slice will cause a panic. The following snippet reworks the previous example to use the `make()` function to initialize the slice:

```
func main() {  
    months := make([]string, 6)  
    ...  
}
```

golang.fyi/ch07/slicemake.go

The `make()` function takes as an argument the type of the slice to be initialized and an initial size for the slice. Then it returns a slice value. In the previous snippet, `make()` does the followings:

- Creates an underlying array of type `[6]string`
- Creates the slice value with length and capacity of 6
- Returns a slice value (not a pointer)

After initialization with the `make()` function, access to a legal index position will return the zero value for the slice element instead of causing a program panic. The `make()` function can take an optional third parameter that specifies the maximum capacity of the slice, as shown in the following example:

```
func main() {  
    months := make([]string, 6, 12)  
    ...  
}
```

golang.fyi/ch07/slicemake2.go

The preceding snippet will initialize the `months` variable with a slice value with an initial length of 6 and a maximum capacity of 12.

Using slices

The simplest operation to do with a slice value is to access its elements. As was mentioned, slices use index notation to access its elements similar to arrays. The following example accesses element at index position 0 and updates to 15:

```
func main () {
    h := []float64{12.5, 18.4, 7.0}
    h[0] = 15
    fmt.Println(h[0])
    ...
}
```

golang.fyi/ch07/slice_use.go

When the program runs, it prints the updated value using index expression `h[0]` to retrieve the value of the item at position 0. Note that the slice expression with only the index number, `h[0]` for instance, returns the value of the item at that position. When, however, the expression includes a colon, say `h[2:]` or `h[:6]`, that expression returns a new slice.

Slice traversal can be done using the traditional `for` statement or with the, more idiomatic, `for...range` statement as shown in the following code snippets:

```
func scale(factor float64, vector []float64) []float64 {
    for i := range vector {
        vector[i] *= factor
    }
    return vector
}

func contains(val float64, numbers []float64) bool {
    for _, num := range numbers {
        if num == val {
            return true
        }
    }
    return false
}
```

golang.fyi/ch07/slice_loop.go

In the previous code snippet, function `scale` uses index variable `i` to update the values in slice `factor` directly, while function `contains` uses the iteration-emitted value stored in `num` to access the slice element. If you need further detail on the `for...range` statement, see Chapter 3, *Go Control Flow*.

Slices as parameters

When a function receives a slice as its parameter, the internal pointer of that slice points to the underlying array of the slice. Therefore, all updates to the slice, within the function, will be seen by the function's caller. For instance, in the following code snippet, all changes to the `vector` parameter will be seen by the caller of function `scale`:

```
func scale(factor float64, vector []float64) {
    for i := range vector {
        vector[i] *= factor
    }
}

golang.fyi/ch07/slice_loop.go
```

Length and capacity

Go provides two built-in functions to query the length and capacity attributes of a slice. Given a slice, its length and maximum capacity can be queried, using the `len` and `cap` functions respectively, as shown in the following example:

```
func main() {
    var vector []float64
    fmt.Println(len(vector)) // prints 0, no panic
    h := make([]float64, 4, 10)
    fmt.Println(len(h), ", ", cap(h))
}
```

Recall that a slice is a value (not a pointer) that has a nil as its zero-value. Therefore, the code is able to query the length (and capacity) of an uninitialized slice without causing a panic at runtime.

Appending to slices

The one indispensable feature of slice types is their ability to dynamically grow. By default, a slice has a static length and capacity. Any attempt to access an index beyond that limit will cause a panic. Go makes available the built-in variadic function `append` to dynamically add new values to a specified slice, growing its lengths and capacity, as necessary. The following code snippet shows how that is done:

```
func main() {
    months := make([]string, 3, 3)
    months = append(months, "Jan", "Feb", "March",
        "Apr", "May", "June")
    months = append(months, []string{"Jul", "Aug", "Sep"}...)
    months = append(months, "Oct", "Nov", "Dec")
    fmt.Println(len(months), cap(months), months)
}
```

golang.fyi/ch07/slice_append.go

The previous snippet starts with a slice with a size and capacity of 3. The `append` function is used to dynamically add new values to the slice beyond its initial size and capacity. Internally, `append` will attempt to fit the appended values within the target slice. If the slice has not been initialized or has an inadequate capacity, `append` will allocate a new underlying array, to store the values of the updated slice.

Copying slices

Recall that assigning or slicing an existing slice value simply creates a new slice value pointing to the same underlying array structure. Go offers the `copy` function, which returns a deep copy of the slice along with a new underlying array. The following snippet shows a `clone()` function, which makes a new copy of a slice of numbers:

```
func clone(v []float64) (result []float64) {
    result = make([]float64, len(v), cap(v))
    copy(result, v)
    return
}
```

golang.fyi/ch07/slice_use.go

In the previous snippet, the `copy` function copies the content of `v` slice into `result`. Both source and target slices must be the same size and of the same type or the `copy` operation will fail.

Strings as slices

Internally, the string type is implemented as a slice using a composite value that points to an underlying array of rune. This affords the string type the same idiomatic treatment given to slices. For instance, the following code snippet uses index expressions to extract slices of strings from a given string value:

```
func main() {
    msg := "Bobsayelloworld!"
    fmt.Println(
        msg[:3], msg[3:7], msg[7:12],
        msg[12:17], msg[len(msg)-1:],
    )
}
```

golang.fyi/ch07/slice_string.go

The slice expression on a string will return a new string value pointing to its underlying array of runes. The string values can be converted to a slice of byte (or slice of rune) as shown in the following function snippet, which sorts the characters of a given string:

```
func sort(str string) string {
    bytes := []byte(str)
    var temp byte
    for i := range bytes {
        for j := i + 1; j < len(bytes); j++ {
            if bytes[j] < bytes[i] {
                temp = bytes[i]
                bytes[i], bytes[j] = bytes[j], temp
            }
        }
    }
    return string(bytes)
}
```

golang.fyi/ch07/slice_string.go

The previous code shows the explicit conversion of a slice of bytes to a string value. Note that each character may be accessed using the index expression.

The map type

The Go map is a composite type that is used as containers for storing unordered elements of the same type indexed by an arbitrary key value. The following code snippet shows a variety of map variables declarations with a variety of key types:

```
var (
    legends map[int]string
    histogram map[string]int
    calibration map[float64]bool
    matrix map[[2][2]int]bool      // map with array key type
    table map[string][]string     // map of string slices

    // map (with struct key) of map of string
    log map[struct{name string}]map[string]string
)
```

golang.fyi/ch07/maptypes.go

The previous code snippet shows several variables declared as maps of different types with a variety of key types. In general, map type is specified as follows:

map[<key_type>]<element_type>

The *key* specifies the type of a value that will be used to index the stored elements of the map. Unlike arrays and slices, map keys can be of any type, not just `int`. Map keys, however, must be of types that are comparable including numeric, string, Boolean, pointers, arrays, struct, and interface types (see Chapter 4, *Data Types*, for discussion on comparable types).

Map initialization

Similar to a slice, a map manages an underlying data structure, opaque to its user, to store its values. An uninitialized map has a nil zero-value as well. Attempts to insert into an uninitialized map will result in a program panic. Unlike a slice, however, it is possible to access elements from a nil map, which will return the zero value of the element.

Like other composite types, maps may be initialized using a composite literal value of the following form:

<map_type>{<comma-separated list of key:value pairs>}

The following snippet shows variable initialization with map composite literals:

```
var (
    histogram map[string]int = map[string]int{
        "Jan":100, "Feb":445, "Mar":514, "Apr":233,
        "May":321, "Jun":644, "Jul":113, "Aug":734,
        "Sep":553, "Oct":344, "Nov":831, "Dec":312,
    }

    table = map[string][]int {
        "Men":[]int{32, 55, 12, 55, 42, 53},
        "Women":[]int{44, 42, 23, 41, 65, 44},
    }
)
```

golang.fyi/ch07/mapinit.go

The literal mapped values are specified using a colon-separated pair of key and value as shown in the previous example. The type of each key and value pair must match that of the declared elements in the map.

Making Maps

Similar to a slice, a map value can also be initialized using the *make* function. Using the *make* function initializes the underlying storage allowing data to be inserted in the map as shown in the following short snippet:

```
func main() {
    hist := make(map[int]string)
    hist["Jan"] = 100
    hist["Feb"] = 445
    hist["Mar"] = 514
    ...
}
```

golang.fyi/ch07/maptypes.go

The *make* function takes as argument the type of the map and it returns an initialized map. In the previous example, the *make* function will initialize a map of type `map[int]string`. The *make* function can optionally take a second parameter to specify the capacity of the map. However, a map will continue to grow as needed ignoring the initial capacity specified.

Using maps

As is done with slice and arrays, index expressions are used to access and update the elements stored in maps. To set or update a map element, use the index expression, on the left side of an assignment, to specify the key of the element to update. The following snippet shows an element with the "Jan" key being updated with the value 100:

```
hist := make(map[int]string)
hist["Jan"] = 100
```

Accessing an element with a given key is done with an index expression, placed on the right side of an assignment, as shown in the following example, where the value indexed with the "Mar" key is assigned the `val` variable:

```
val := hist["Mar"]
```

Earlier it was mentioned that accessing a non-existent key will return the zero-value for that element. For instance, the previous code would return 0 if the element with the key "Mar" does not exist in the map. As you can imagine, this can be a problem. How would you know whether you are getting an actual value or the zero-value? Fortunately, Go provides a way to explicitly test for the absence of an element by returning an optional Boolean value as part of the result of an index expression, as shown in the following snippet:

```
func save(store map[string]int, key string, value int) {
    val, ok := store[key]
    if !ok {
        store[key] = value
    }else{
        panic(fmt.Sprintf("Slot %d taken", val))
    }
}
```

golang.fyi/ch07/map_use.go

The function in the preceding snippet tests the existence of a key before updating its value. Called the *comma-ok* idiom, the Boolean value stored in the `ok` variable is set to false when the value is not actually found. This allows the code to distinguish between the absence of a key and the zero value of the element.

Map traversal

The `for...range` loop statement can be used to walk the content of a map value. The `range` expression emits values for both key and element values with each iteration. The following code snippet shows the traversal of map `hist`:

```
for key, val := range hist {  
    adjVal := int(float64(val) * 0.100)  
    fmt.Printf("%s (%d):", key, val)  
    for i := 0; i < adjVal; i++ {  
        fmt.Print(".")  
    }  
    fmt.Println()  
}
```

golang.fyi/ch07/map_use.go

Each iteration returns a key and its associated element value. Iteration order, however, is not guaranteed. The internal map iterator may traverse the map in a different order with each run of the program. In order to maintain a predictable traversal order, keep (or generate) a copy of the keys in a separate structure, such as a slice for instance. During traversal, range over the slice of keys to traverse in a predictable manner.



You should be aware that update done to the emitted value during the iteration will be lost. Instead, use an index expression, such as `hist[key]` to update an element during iteration. For details on `for...range` loop, refer to Chapter 3, *Go Control Flow*, for a thorough explanation of Go `for` loops.

Map functions

Besides the `make` function, discussed earlier, map types support two additional functions discussed in the following table:

Function	Description
<code>len(map)</code>	As with other composite types, the built-in <code>len()</code> function returns the number of entries in a map. For instance, the following would print 3: <code>h := map[int]bool{3:true, 7:false, 9:false}</code> <code>fmt.Println(len(h))</code> The <code>len</code> function will return zero for an uninitialized map.
<code>delete(map, key)</code>	The built-in <code>delete</code> function deletes an element from a given map associated with the provided key. The following code snippet would print 2: <code>h := map[int]bool{3:true, 7:false, 9:false}</code> <code>delete(h, 7)</code> <code>fmt.Println(len(h))</code>

Maps as parameters

Because a map maintains an internal pointer to its backing storage structure, all updates to map parameter within a called function will be seen by the caller once the function returns. The following sample shows a call to the `remove` function to change the content of a map. The passed variable, `hist`, will reflect the change once the `remove` function returns:

```
func main() {
    hist := make(map[string]int)
    hist["Jun"] = 644
    hist["Jul"] = 113
    remove(hist, "Jun")
    len(hist) // returns 1
}
func remove(store map[string]int, key string) error {
    _, ok := store[key]
    if !ok {
        return fmt.Errorf("Key not found")
    }
    delete(store, key)
    return nil
}
```

golang.fyi/ch07/map_use.go

The struct type

The last type discussed in this chapter is Go's `struct`. It is a composite type that serves as a container for other named types known as fields. The following code snippet shows several variables declared as structs:

```
var(
    empty struct{}
    car struct{make, model string}
    currency struct{name, country string; code int}
    node struct{
        edges []string
        weight int
    }
    person struct{
        name string
        address struct{
            street string
            city, state string
            postal string
        }
    }
)
```

golang.fyi/ch07/structtypes.go

Note that the struct type has the following general format:

`struct{<field declaration set>}`

The `struct` type is constructed by specifying the keyword `struct` followed by a set of field declarations enclosed within curly brackets. In its most common form, a field is a unique identifier with an assigned type which follows Go's variable declaration conventions as shown in the previous code snippet (`struct` also support anonymous fields, covered later).

It is crucial to understand that the type definition for a `struct` includes all of its declared fields. For instance, the type for the `person` variable (see earlier code snippet) is the entire set of fields in the declaration `struct { name string; address struct { street string; city string; state string; postal string } }`. Therefore, any variable or expression requiring that type must repeat that long declaration. We will see later how that is mitigated by using named types for `struct`.

Accessing struct fields

A struct uses a *selector expression* (or dot notation) to access the values stored in fields. For instance, the following would print the value of the `name` field of the `person` struct variable from the previous code snippet:

```
fmt.Println(person.name)
```

Selectors can be chained to access fields that are nested inside a struct. The following snippet would print the street and city for the nested address value of a `person` variable:

```
fmt.Println(person.address.street)
fmt.Println(person.address.city)
```

Struct initialization

Similar to arrays, structs are pure values with no additional underlying storage structure. The fields for an uninitialized struct are assigned their respective zero values. This means an uninitialized struct requires no further allocation and is ready to be used.

Nevertheless, a struct variable can be explicitly initialized using a composite literal of the following form:

<struct_type>{<positional or named field values>}

The composite literal value for a struct can be initialized by a set of field values specified by their respective positions. Using this approach, all field values must be provided, to match their respective declared types, as shown in the following snippet:

```
var (
    currency = struct{
        name, country string
        code int
    }{
        "USD", "United States",
        840,
    }
    ...
)
```

golang.fyi/ch07/structinit.go

In the previous struct literal, all field values of the struct are provided, matching their declared field types. Alternatively, the composite literal value of a struct can be specified using a field indices and their associated value. As before, the index (the field name) and its value is separated by a colon, as shown in the following snippet:

```
var(
    car = struct{make, model string}{make:"Ford", model:"F150"}
    node = struct{
        edges []string
        weight int
    }{
        edges: []string{"north", "south", "west"},
    }
    ...
)
```

golang.tutorialspoint.com/structinit.go

As you can see, field values of the composite literal can be selectively specified when the index and its value are provided. For instance, in the initialization of the `node` variable, the `edges` field is initialized while `weight` is omitted.

Declaring named struct types

Attempting to reuse struct types can get unwieldy fast. For instance, having to write `struct { name string; address struct { street string; city string; state string; postal string } }` to express a struct type, every time it is needed, would not scale, would be error prone, and would make for grumpy Go developers. Luckily, the proper idiom to fix this is to use named types, as illustrated in the following source code snippet:

```
type person struct {
    name      string
    address   address
}

type address struct {
    street      string
    city, state string
    postal      string
}

func makePerson() person {
    addr := address{
```

```
        city: "Goville",
        state: "Go",
        postal: "12345",
    }
    return person{
        name: "vladimir vivien",
        address: addr,
    }
}
```

golang.fyi/ch07/structtype_dec.go

The previous example binds struct type definitions to the identifiers `person` and `address`. This allows the struct types to be reused in different contexts without the need to carry around the long form of the type definitions. You can refer to Chapter 4, *Data Types*, to learn more about named types.

The anonymous field

Previous definitions of struct types involved the use of named fields. However, it is also possible to define a field with only its type, omitting the identifier. This is known as an anonymous field. It has the effect of embedding the type directly into the struct.

This concept is demonstrated in the following code snippet. Both `diameter` and the `name`, are embedded as anonymous fields in the `planet` type:

```
type diameter int

type name struct {
    long    string
    short   string
    symbol  rune
}

type planet struct {
    diameter
    name
    desc string
}

func main() {
    earth := planet{
        diameter: 7926,
        name: name{
            long:    "Earth",
            short:   "E",
        }
    }
}
```

```
        symbol: '\u2641',
    },
    desc: "Third rock from the Sun",
}
...
}
```

golang.fyi/ch07/struct_embed.go

The `main` function in the previous snippet shows how the anonymous fields are accessed and updated, as is done in the `planet` struct. Notice the names of the embedded types become the field identifiers in the composite literal value for the struct.

To simplify field name resolution, Go follows the following rules when using anonymous fields:

- The name of the type becomes the name of the field
- The name of an anonymous field may not clash with other field names
- Use only the unqualified (omit package) type name of imported types

These rules also hold when accessing the fields of embedded structs directly using selector expressions, as is shown in the following code snippet. Notice the name of the embedded types are resolved as fields names:

```
func main() {
    jupiter := planet{}
    jupiter.diameter = 88846
    jupiter.name.long = "Jupiter"
    jupiter.name.short = "J"
    jupiter.name.symbol = '\u2643'
    jupiter.desc = "A ball of gas"
    ...
}
```

golang.fyi/ch07/struct_embed.go

Promoted fields

Fields of an embedded struct can be *promoted* to its enclosing type. Promoted fields appear in selector expressions without the qualified name of their types, as shown in the following example:

```
func main() {
    ...
}
```

```
saturn := planet{}
saturn.diameter = 120536
saturn.long = "Saturn"
saturn.short = "S"
saturn.symbol = '\u2644'
saturn.desc = "Slow mover"
...
}
```

golang.fyi/ch07/struct_embed.go

In the previous snippet, the highlighted fields are promoted from the embedded type name by omitting it from the selector expression. The values of the fields `long`, `short`, and `symbol` come from embedded type `name`. Again, this will only work if the promotion does not cause any identifier clashes. In case of ambiguity, the fully qualified selector expression can be used.

Structs as parameters

Recall that struct variables store actual values. This implies that a new copy of a struct value is created whenever a `struct` variable is reassigned or passed in as a function parameter. For instance, the following will not update the value of `name` after the call to `updateName()`:

```
type person struct {
    name    string
    title   string
}
func updateName(p person, name string) {
    p.name = name
}

func main() {
    p := person{}
    p.name = "unknown"
    ...
    updateName(p, "Vladimir Vivien")
}
```

golang.fyi/ch07/struct_ptr.go

This can be remedied by passing a pointer to the `struct` value of the `person` type, as shown in the following snippet:

```
type person struct {
    name     string
    title   string
}

func updateName(p *person, name string) {
    p.name = name
}

func main() {
    p := new(person)
    p.name = "unknown"
    ...
    updateName(p, "Vladimir Vivien")
}
```

golang.fyi/ch07/struct_ptr2.go

In this version, the `p` variable is declared as `*person` and is initialized using the built-in `new()` function. After `updateName()` returns, its changes are seen by the calling function.

Field tags

The last topic on structs has to do with field tags. During the definition of a `struct` type, optional `string` values may be added to each field declaration. The value of the `string` is arbitrary and it can serve as hints to tools or other APIs that use reflection to consume the tags.

The following shows a definition of the `Person` and `Address` structs that are tagged with JSON annotation which can be interpreted by Go's JSON encoder and decoder (found in the standard library):

```
type Person struct {
    Name     string `json:"person_name"`
    Title   string `json:"person_title"`
    Address `json:"person_address_obj"`
}

type Address struct {
    Street string `json:"person_addr_street"`
    City   string `json:"person_city"`
    State  string `json:"person_state"`
}
```

```
    Postal string `json:"person_postal_code"`
}
func main() {
    p := Person{
        Name: "Vladimir Vivien",
        Title : "Author",
        ...
    }
    ...
    b, _ := json.Marshal(p)
    fmt.Println(string(b))
}
```

golang.fyi/ch07/struct_ptr2.go

Notice the tags are represented as raw string values (wrapped within a pair of ``). The tags are ignored by normal code execution. However, they can be collected using Go's reflection API as is done by the JSON library. You will encounter more on this subject in Chapter 10, *Data IO in Go*, when the book discusses input and output streams.

Summary

This chapter covered a lot of ground as it walked through each of the composite types found in Go to provide insightful coverage of their characteristics. The chapter opened with a coverage of the array type, where readers learned how to declare, initialize, and use array values. Next, readers learned all about the slice type, specifically the declaration, initialization, and practical examples that uses slice index expressions to create new or re-slice existing slices. The chapter covered the map type, which included information on map initialization, access, update, and traversal. Lastly, the chapter provided information about the definition, initialization, and usage of the struct type.

Needless to say, this is probably one of the longest chapters of the book. However, the information covered here will prove to be invaluable as the book continues to explore new topics. The next chapter will introduce the idea of using Go to support object-like idioms using methods and interfaces.

8

Methods, Interfaces, and Objects

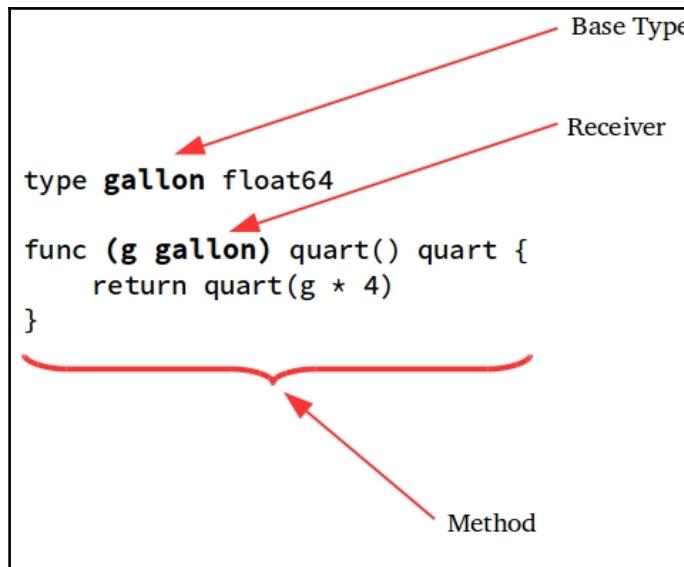
Using your skills at this point, you can write an effective Go program using the fundamental concepts covered so far. As you will see in this chapter, the Go type system can support idioms that go beyond simple functions. While the designers of Go did not intend to create an object-oriented language with deep class hierarchies, the language is perfectly capable of supporting type compositions with advanced features to express the creation of complex object-like structures, as covered in the following topics:

- Go methods
- Objects in Go
- The interface type
- Type assertion

Go methods

A Go function can be defined with a scope narrowed to that of a specific type. When a function is scoped to a type, or attached to the type, it is known as a *method*. A method is defined just like any other Go function. However, its definition includes a *method receiver*, which is an extra parameter placed before the method's name, used to specify the host type to which the method is attached.

To better illustrate this concept, the following figure highlights the different parts involved in defining a method. It shows the `quart` method attached to the `type gallon` based receiver via the `g` `gallon` receiver parameter:



As mentioned, a method has the scope of a type. Therefore, it can only be accessed via a declared value (concrete or pointer) of the attached type using *dot notation*. The following program shows how the declared method `quart` is accessed using this notation:

```
package main
import "fmt"

type gallon float64

func (g gallon) quart() float64 {
    return float64(g * 4)
}
func main() {
    gal := gallon(5)
    fmt.Println(gal.quart())
}
```

golang.fyi/ch08/method_basic.go

In the previous example, the `gal` variable is initialized as the `gallon` type. Therefore, the `quart` method can be accessed using `gal.quart()`.

At runtime, the receiver parameter provides access to the value assigned to the base type of the method. In the example, the `quart` method receives the `g` parameter, which passes in a copy of the value for the declared type. So when the `gal` variable is initialized with a value of 5, a call to `gal.quart()` sets the receiver parameter `g` to 5. So the following would then print a value of 20:

```
func main() {
    gal := gallon(5)
    fmt.Println(gal.quart())
}
```

It is important to note that the base type for method receivers cannot be a pointer (nor an interface). For instance, the following will not compile:

```
type gallon *float64
func (g gallon) quart() float64 {
    return float64(g * 4)
}
```

The following shows a lengthier version of the source that implements a more general liquid volume conversion program. Each volumetric type receives its respective methods to expose behaviors attributed to that type:

```
package main
import "fmt"

type ounce float64
func (o ounce) cup() cup {
    return cup(o * 0.1250)
}

type cup float64
func (c cup) quart() quart {
    return quart(c * 0.25)
}
func (c cup) ounce() ounce {
    return ounce(c * 8.0)
}

type quart float64
func (q quart) gallon() gallon {
    return gallon(q * 0.25)
}
func (q quart) cup() cup {
```

```
    return cup(q * 4.0)
}

type gallon float64
func (g gallon) quart() quart {
    return quart(g * 4)
}

func main() {
    gal := gallon(5)
    fmt.Printf("%.2f gallons = %.2f quarts\n", gal, gal.quart())
    ozs := gal.quart().cup().ounce()
    fmt.Printf("%.2f gallons = %.2f ounces\n", gal, ozs)
}
```

github.com/vladimirvivien/learning-go/ch08/methods.go

For instance, converting 5 gallons to ounces can be done by invoking the proper conversion methods on a given value, as follows:

```
gal := gallon(5)
ozs := gal.quart().cup().ounce()
```

The entire implementation uses a simple, but effective, typical structure to represent both data type and behavior. Reading the code, it cleanly expresses its intended meaning without any reliance on heavy class structures.



Method set The number of methods attached to a type, via the receiver parameter, is known as the type's *method set*. This includes both concrete and pointer value receivers. The concept of a method set is important in determining type equality, interface implementation, and support of the notion of the empty method set for the *empty interface* (all discussed in this chapter).

Value and pointer receivers

One aspect of methods that has escaped discussion so far is that receivers are normal function parameters. Therefore, they follow the pass-by-value mechanism of Go functions. This implies that the invoked method gets a copy of the original value from the declared type.

Receiver parameters can be passed as either values or pointers of the base type. For instance, the following program shows two methods, `half` and `double`; both directly update the value of their respective method receiver parameters, `g`:

```
package main
import "fmt"
type gallon float64
func (g gallon) quart() float64 {
    return float64(g * 4)
}
func (g gallon) half() {
    g = gallon(g * 0.5)
}
func (*gallon) double() {
    *g = gallon(*g * 2)
}
func main() {
    var gal gallon = 5
    gal.half()
    fmt.Println(gal)
    gal.double()
    fmt.Println(gal)
}
```

golang.fyi/ch08/receiver_ptr.go

In the `half` method, the code updates the receiver parameter with `g = gallon(g * 0.5)`. As you would expect, this will not update the original declared value, but rather the copy stored in the `g` parameter. So, when `gal.half()` is invoked in `main`, the original value remains unchanged and the following would print 5:

```
func main() {
    var gal gallon = 5
    gal.half()
    fmt.Println(gal)
}
```

Similar to regular function parameters, a receiver parameter that uses a pointer to refer to its base value allows the code to dereference the original value to update it. This is highlighted in the `double` method following snippet. It uses a method receiver of the `*gallon` type, which is updated using `*g = gallon(*g * 2)`. So when the following is invoked in `main`, it would print a value of **10**:

```
func main() {
    var gal gallon = 5
    gal.double()
    fmt.Println(gal)
}
```

Pointer receiver parameters are widely used in Go. This is because they make it possible to express object-like primitives that can carry both state and behaviors. As the next section shows, pointer receivers, along with other type features, are the basis for creating objects in Go.

Objects in Go

The lengthy introductory material from the previous sections was the setup to lead to the discussion of objects in Go. It has been mentioned that Go was not designed to function as traditional object-oriented language. There are no object or class keywords defined in Go. So then, why are we discussing objects in Go at all? Well, it turns out that Go perfectly supports object idioms and the practice of object-oriented programming without the heavy baggage of classical hierarchies and complex inheritance structures found in other object-oriented languages.

Let us review some of the primordial features usually attributed to an object-oriented language in the following table.

Object feature	Go	Comment
Object: A data type that stores states and exposes behavior	Yes	In Go all types can achieve this. There is no special type called a class or object to do this. Any type can receive a set of method to define its behavior, although the <code>struct</code> type comes the closest to what is commonly called an object in other languages.
Composition	Yes	Using a type such as a <code>struct</code> or an <code>interface</code> (discussed later), it is possible to create objects and express their polymorphic relationships through composition.

Subtype via interface	Yes	A type that defines a set of behaviors (methods) that other types may implement. Later you will see how it is used to implement object sub-typing.
Modularity and encapsulation	Yes	Go supports physical and logical modularity at its core with concepts such packages and an extensible type system, and code element visibility.
Type inheritance	No	Go does not support polymorphism through inheritance. A newly declared named type does not inherit all attributes of its underlying type and are treated differently by the type system. As a consequence, it is hard to implement inheritance via type lineage as found in other languages.
Classes	No	There is no notion of a class type that serves as the basis for objects in Go. Any data type in Go can be used as an object.

As the previous table suggests, Go supports the majority of concepts that are usually attributed to object-oriented programming. The remainder of this chapter covers topics and examples showing how to use Go as an object-oriented programming language.

The struct as object

Nearly all Go types can play the role of an object by storing states and exposing methods that are capable of accessing and modifying those states. The `struct` type, however, offers all of the features that are traditionally attributed to objects in other languages, such as:

- Ability to host methods
- Ability to be extended via composition
- Ability to be sub-typed (with help from the Go `interface` type)

The remainder of the chapter will base its discussion of objects on using the `struct` type.

Object composition

Let us start with the following simple example to demonstrate how the `struct` type may be used as an object that can achieve polymorphic composition. The following source code snippet implements a typical structure that models components of motorized transportation including fuel, engine, vehicle, truck, and plane:

```
type fuel int
const (
    GASOLINE fuel = iota
    BIO
    ELECTRIC
    JET
)
type vehicle struct {
    make string
    model string
}

type engine struct {
    fuel fuel
    thrust int
}
func (e *engine) start() {
    fmt.Println ("Engine started.")
}

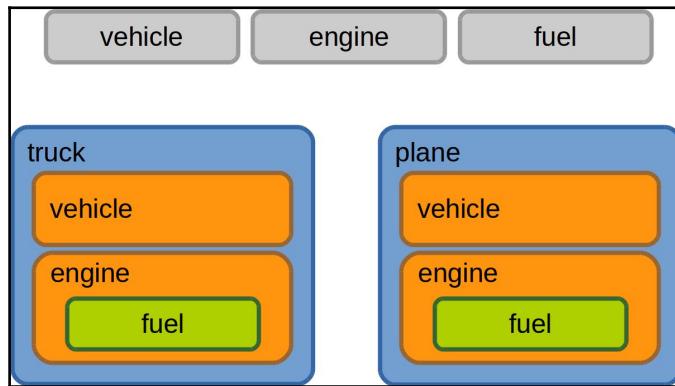
type truck struct {
    vehicle
    engine
    axels int
    wheels int
    class int
}
func (t *truck) drive() {
    fmt.Printf("Truck %s %s, on the go!\n", t.make, t.model)
}

type plane struct {
    vehicle
    engine
    engineCount int
    fixedWings bool
    maxAltitude int
}
func (p *plane) fly() {
    fmt.Printf(
```

```
    "Aircraft %s %s clear for takeoff!\n",
    p.make, p.model,
)
}
```

golang.fyi/ch08/structobj.go

The components and their relationships declared in the previous code snippet are illustrated in the following figure to visualize the type mapping and their compositions:



Go uses the *composition over inheritance* principle to achieve polymorphism using the type embedding mechanism supported by the `struct` type. In Go, there is no support for polymorphism via type inheritance. Recall that each type is independent and is considered to be different from all others. In fact, the semantics in the model above is slightly broken. Types `truck` and `plane` are shown to be composed of (or has-a) the `vehicle` type, which does not sound correct. Instead, the proper, or at least a more correct, representation would be to show that the types `truck` and `plane` *is a* `vehicle` via a subtype relationship. Later in the chapter, we will see how this can be achieved using the `interface` type.

Field and method promotion

Now that the objects have been established in the previous section, let us spend some time discussing the visibility of fields, methods, and embedded types inside the structs. The following source snippet shows a continuation of the previous example. It declares and initializes a variable `t` of type `truck` and `p` for `plane`. The former is initialized using a struct literal and the latter is updated using dot notation:

```
func main() {
    t := &truck {
        vehicle:vehicle{"Ford", "F750"},
        engine:engine{GASOLINE+BIO, 700},
        axels:2,
        wheels:6,
        class:3,
    }
    t.start()
    t.drive()

    p := &plane{}
    p.make = "HondaJet"
    p.model = "HA-420"
    p.fuel = JET
    p.thrust = 2050
    p.engineCount = 2
    p.fixedWings = true
    p.maxAltitude = 43000
    p.start()
    p.fly()
}
```

golang.fyi/ch08/structobj.go

One of the more interesting details in the previous snippet is how the `struct` type embedding mechanism promotes fields and methods when accessed using dot notation. For instance, the following fields (`make`, `model`, `fuel`, and `thrust`), are all declared in types that are embedded inside of the `plane` type:

```
p.make = "HondaJet"
p.model = "HA-420"
p.fuel = JET
p.thrust = 2050
```

The previous fields are promoted from their embedded types. They are accessed as if they are members of the `plane` type when, in fact, they are coming from the types `vehicle` and `engine` respectively. To avoid ambiguity, the name of the fields can be qualified as shown here:

```
p.vehicle.make = "HondaJet"  
p.vehicle.model = "HA-420"  
p.engine.fuel = JET  
p.engine.thrust = 2050
```

Methods can also be promoted in a similar way. For instance, in the previous code we saw the methods `t.start()` and `p.start()` being invoked. However, neither type, `truck` nor `plane`, are receivers of a method named `start()`. As shown in the program from earlier, the `start()` method is defined for the `engine` type. Since the `engine` type is embedded in the types `truck` and `plane`, the `start()` method is promoted in scope to these enclosing types and is therefore accessible.

The constructor function

Since Go does not support classes, there is no such concept as a constructor. However, one conventional idiom you will encounter in Go is the use of a factory function to create and initialize values for a type. The following snippet shows a portion of the previous example that has been updated to use a constructor function for creating new values of the `plane` and `truck` types:

```
type truck struct {  
    vehicle  
    engine  
    axels int  
    wheels int  
    class int  
}  
func newTruck(mk, mdl string) *truck {  
    return &truck {vehicle:vehicle{mk, mdl}}  
}  
  
type plane struct {  
    vehicle  
    engine  
    engineCount int  
    fixedWings bool  
    maxAltitude int  
}  
func newPlane(mk, mdl string) *plane {
```

```
p := &plane{}  
p.make = mk  
p.model = mdl  
return p  
}
```

golang.fyi/ch08/structobj2.go

While not required, providing a function to help with the initialization of composite values, such as a struct, increases the usability of the code. It provides a place to encapsulate repeatable initialization logic that can enforce validation requirements. In the previous example, both constructor functions, `newTruck` and `newPlane`, are passed the make and model information to create and initialize their respected values.

The interface type

When you talk to people who have been doing Go for a while, they almost always list the interface as one of their favorite features of the language. The concept of interfaces in Go, similar to other languages, such as Java, is a set of methods that serves as a template to describe behavior. A Go interface, however, is a type specified by the `interface{ }` literal, which is used to list a set of methods that satisfies the interface. The following example shows the `shape` variable being declared as an interface:

```
var shape interface {  
    area() float64  
    perim() float64  
}
```

In the previous snippet, the `shape` variable is declared and assigned an unnamed type, `interface{area()float64; perim()float64}`. Declaring variables with unnamed interface literal types is not really practical. Using idiomatic Go, an interface type is almost always declared as a named type. The previous snippet can be rewritten to use a named interface type, as shown in the following example:

```
type shape interface {  
    area() float64  
    perim() float64  
}  
var s shape
```

Implementing an interface

The interesting aspect of interfaces in Go is how they are implemented and ultimately used. Implementing a Go interface is done implicitly. There is no separate element or keyword required to indicate the intent of implementation. Any type that defines the method set of an interface type automatically satisfies its implementation.

The following source code shows the `rect` type as an implementation of the `interface` type `shape`. The `rect` type is defined as a `struct` with receiver methods `area` and `perim`. This fact automatically qualifies `rect` as an implementation of `shape`:

```
type shape interface {
    area() float64
    perim() float64
}

type rect struct {
    name string
    length, height float64
}

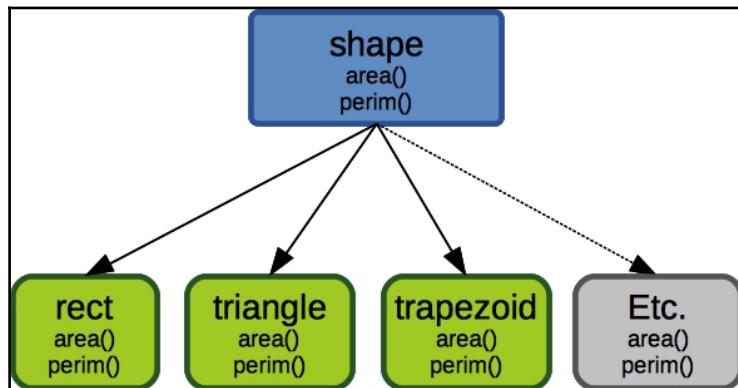
func (r *rect) area() float64 {
    return r.length * r.height
}

func (r *rect) perim() float64 {
    return 2*r.length + 2*r.height
}
```

golang.fyi/ch08/interface_impl.go

Subtyping with Go interfaces

Earlier, during the discussion on objects, it was mentioned that Go favors composition (*has-a*) relationships when building objects. While that is true, Go can also express "is-a" relationships among objects using subtyping via interfaces. In our previous example, it can be argued that the `rect` type (and any other type that implements the methods `area()` and `perim()`) can be treated as a subtype of `shape`, as shown in the following figure:



As you may expect, any subtype of `shape` can participate in expressions or be passed as functions (or methods) parameters where the `shape` type is expected. This is shown in the following code snippet where both types, `rect` (defined previously) and `triangle`, are able to be passed to the `shapeInfo(shape)` function to return a `string` value containing shape calculations:

```
type triangle struct {
    name string
    a, b, c float64
}

func (t *triangle) area() float64 {
    return 0.5*(t.a * t.b)
}

func (t *triangle) perim() float64 {
    return t.a + t.b + math.Sqrt((t.a*t.a) + (t.b*t.b))
}

func (t *triangle) String() string {
    return fmt.Sprintf(
        "%s[sides: a=%.2f b=%.2f c=%.2f]", t.name, t.a, t.b, t.c,
```

```
)  
}  
func shapeInfo(s shape) string {  
    return fmt.Sprintf(  
        "Area = %.2f, Perim = %.2f",  
        s.area(), s.perim(),  
    )  
}  
  
func main() {  
    r := & rect{"Square", 4.0, 4.0}  
    fmt.Println(r, "=>", shapeInfo(r))  
  
    t := & triangle{"Right Triangle", 1, 2, 3}  
    fmt.Println(t, "=>", shapeInfo(t))  
}
```

golang.fyi/ch08/interface_impl.go

Implementing multiple interfaces

The implicit mechanism of interfaces allows any named type to satisfy multiple interface types at once. This is achieved simply by having the method set of a given type intersect with the methods of each `interface` type to be implemented. Let us re-implement the previous code to show how this is done. Two new interfaces are introduced, `polygon` and `curved`, to better capture and categorize information and the behavior of shapes, as shown in the following code snippet:

```
type shape interface {  
    area() float64  
}  
  
type polygon interface {  
    perim()  
}  
  
type curved interface {  
    circonf()  
}  
type rect struct {...}  
func (r *rect) area() float64 {  
    return r.length * r.height  
}  
func (r *rect) perim() float64 {  
    return 2*r.length + 2*r.height
```

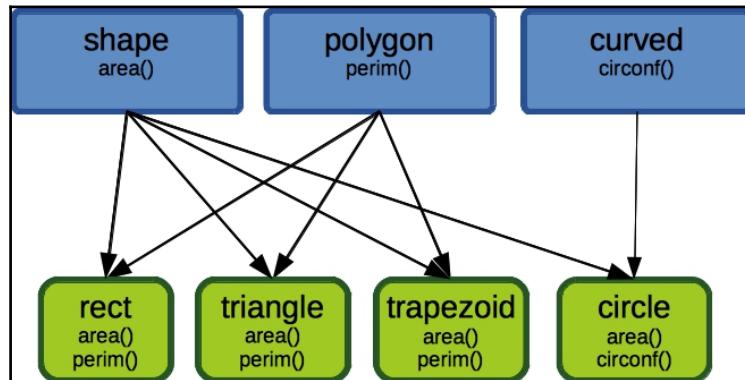
```
}

type triangle struct {...}
func (t *triangle) area() float64 {
    return 0.5*(t.a * t.b)
}
func (t *triangle) perim() float64 {
    return t.a + t.b + math.Sqrt((t.a*t.a) + (t.b*t.b))
}

type circle struct { ... }
func (c *circle) area() float64 {
    return math.Pi * (c.rad*c.rad)
}
func (c *circle) circonf() float64 {
    return 2 * math.Pi * c.rad
}
```

golang.fyi/ch08/interface_impl2.go

The previous source code snippet shows how types can automatically satisfy multiple interfaces by simply declaring methods that satisfy the interfaces' method sets. This is illustrated by the following figure:



Interface embedding

Another interesting aspect of the `interface` type is its support for type embedding (similar to the `struct` type). This gives you the flexibility to structure your types in ways that maximize type reuse. Continuing with the shape example, the following code snippet reorganizes and reduces the previous interface count from three to two by embedding `shape` into the other two types:

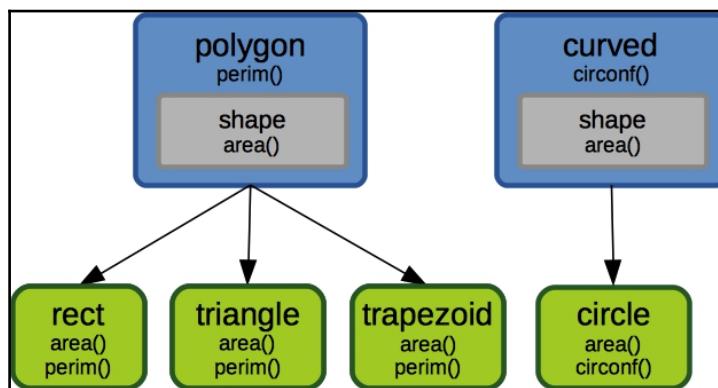
```
type shape interface {
    area() float64
}

type polygon interface {
    shape
    perim()
}

type curved interface {
    shape
    circonf()
}
```

golang.fyi/ch08/interface_impl3.go

The following illustration shows how the interface types may be combined so the *is-a* relationship still satisfies the relationships between code components:



When embedding interface types, the enclosing type will inherit the method set of the embedded types. The compiler will complain if the embedded type causes method signatures to clash. Embedding becomes a crucial feature, especially when the code applies type validation using type checking. It allows a type to roll up type information, thus reducing unnecessary assertion steps (type assertion is discussed later).

The empty interface type

The `interface{}` type, or the empty interface type, is the literal representation of an interface type with an empty method set. According to our discussion so far, it can be deduced that *all types implement the empty interface* since all types can have a method set with zero or more members.

When a variable is assigned the `interface{}` type, the compiler relaxes its build-time type checks. The variable, however, still carries type information that can be queried at runtime. The following code illustrates how this works:

```
func main() {
    var anyType interface{}
    anyType = 77.0
    anyType = "I am a string now"
    fmt.Println(anyType)

    printAnyType("The car is slow")
    m := map[string] string{"ID":"12345", "name":"Kerry"}
    printAnyType(m)
    printAnyType(1253443455)
}

func printAnyType(val interface{}) {
    fmt.Println(val)
}
```

golang.fyi/ch08/interface_empty.go

In the previous code, the `anyType` variable is declared to be of the type `interface{}`. It is able to be assigned values of different types without complaints from the compiler:

```
anyType = 77.0
anyType = "I am a string now"
```

The `printAnyType()` function takes a parameter of the type `interface{}`. This means the function can be passed the values of any valid type, as shown here:

```
printAnyType("The car is slow")
m := map[string] string{"ID":"12345", "name":"Kerry"}
printAnyType(m)
printAnyType(1253443455)
```

The empty interface is crucially important for idiomatic Go. Delaying type-checking until runtime makes the language feels more dynamic without completely sacrificing strong typing. Go offers mechanisms such as type assertion (covered next) to query the type information carried by interfaces at runtime.

Type assertion

When an interface (empty or otherwise) is assigned to a variable, it carries type information that can be queried at runtime. Type assertion is a mechanism that is available in Go to idiomatically narrow a variable (of `interface` type) down to a concrete type and value that are stored in the variable. The following example uses type assertion in the `eat` function to select which `food` type to select in the `eat` function:

```
type food interface {
    eat()
}

type veggie string
func (v veggie) eat() {
    fmt.Println("Eating", v)
}

type meat string
func (m meat) eat() {
    fmt.Println("Eating tasty", m)
}

func eat(f food) {
    veg, ok := f.(veggie)
    if ok {
        if veg == "okra" {
            fmt.Println("Yuk! not eating ", veg)
        }else{
            veg.eat()
        }
    }
}
```

```
        return
    }

    mt, ok := f.(meat)
    if ok {
        if mt == "beef" {
            fmt.Println("Yuk! not eating ", mt)
        }else{
            mt.eat()
        }
        return
    }

    fmt.Println("Not eating whatever that is: ", f)
}
```

golang.fyi/interface_assert.go

The `eat` function takes the `food` interface type as its parameter. The code shows how to use idiomatic Go to extract the static type and value stored in the `f` interface parameter using assertion. The general form for type assertion expression is given as follows:

<interface_variable>.(<concrete type name>)

The expression starts with the variable of the interface type. It is then followed by a dot and the concrete type being asserted enclosed in parentheses. The type assertion expression can return two values: one is the concrete value (extracted from the interface) and the second is a Boolean indicating the success of the assertion, as shown here:

value, boolean := <interface_variable>.(<concrete type name>)

This is the form of assertion that is shown in the following snippet (extracted from the earlier example) when narrowing the `f` parameter to a specific type of `food`. If the type is asserted to be `meat`, then the code continues to test the value of the `mt` variable:

```
mt, ok := f.(meat)
if ok {
    if mt == "beef" {
        fmt.Println("Yuk! not eating ", mt)
    }else{
        mt.eat()
    }
    return
}
```

A type assertion expression can also return just the value, as follows:

value := <interface_variable>.(<concrete type name>)

This form of assertion is risky to do as the runtime will cause a panic in the program if the value stored in the interface variable is not of the asserted type. Use this form only if you have other safeguards to either prevent or gracefully handle a panic.

Lastly, when your code requires multiple assertions to test many types at runtime, a much nicer idiom for assertions is the type `switch` statement. It uses the `switch` statement semantic to query static type information from an interface value using case clauses. The `eat` function from the previous food-related example can been updated to use a type `switch` instead of `if` statement, as shown in the following code snippet:

```
func eat(f food) {
    switch morsel := f.(type) {
    case veggie:
        if morsel == "okra" {
            fmt.Println("Yuk! not eating ", morsel)
        } else{
            morsel.eat()
        }
    case meat:
        if morsel == "beef" {
            fmt.Println("Yuk! not eating ", morsel)
        } else{
            morsel.eat()
        }
    default:
        fmt.Println("Not eating whatever that is: ", f)
    }
}
```

golang.fyi/interface_assert2.go

Notice the code is much nicer to read. It can support any number of cases and is clearly laid out with visual clues that makes it easy to reason about. The `switch` type also makes the panic issue go away by simply specifying a default case that can handle any types not specifically handled in the case clause.

Summary

This chapter attempted to give a broad and, at the same, somewhat comprehensive view of several important topics including methods, interfaces, and objects in Go. The chapter started with coverage of attaching methods to types using receiver parameters. Next the reader was introduced to objects and how to create idiomatic object-based programming in Go. Lastly, the chapter presented a comprehensive overview of the interface type and how it is used to support object semantics in Go. The next chapter takes the reader through one of the most fundamental concepts that has made Go such a sensation among developers: concurrency!

9

Concurrency

Concurrency is considered to be the one of the most attractive features of Go. Adopters of the language revel in the simplicity of its primitives to express correct concurrency implementations without the pitfalls that usually come with such endeavors. This chapter covers the necessary topics to understand and create concurrent Go programs, including the following:

- Goroutines
- Channels
- Writing concurrent programs
- The sync package
- Detecting race conditions
- Parallelism in Go

Goroutines

If you have worked in other languages, such as Java or C/C++, you are probably familiar with the notion of concurrency. It is the ability of a program to run two or more paths of execution independently. This is usually done by exposing a thread primitive directly to the programmer to create and manage concurrency.

Go has its own concurrency primitive called the *goroutine*, which allows a program to launch a function (routine) to execute independently from its calling function. Goroutines are lightweight execution contexts that are multiplexed among a small number of OS-backed threads and scheduled by Go's runtime scheduler. That makes them cheap to create without the overhead requirements of true kernel threads. As such, a Go program can initiate thousands (even hundreds of thousands) of goroutines with minimal impact on performance and resource degradation.

The go statement

Goroutines are launched using the `go` statement as follows:

`go <function or expression>`

A goroutine is created with the `go` keyword followed by the function to schedule for execution. The specified function can be an existing function, an anonymous function, or an expression that calls a function. The following code snippet shows an example of the use of goroutines:

```
func main() {
    go count(10, 50, 10)
    go count(60, 100, 10)
    go count(110, 200, 20)
}
func count(start, stop, delta int) {
    for i := start; i <= stop; i += delta {
        fmt.Println(i)
    }
}
```

golang.fyi/ch09/goroutine0.go

In the previous code sample, when the `go count()` statement is encountered in the `main` function, it launches the `count` function in an independent execution context. Both the `main` and `count` functions will be executing concurrently. As a side effect, `main` will complete before any of the `count` functions get a chance to print anything to the console.

Later in the chapter, we will see how to handle synchronization idiomatically between goroutines. For now, let us use `fmt.Scanln()` to block and wait for keyboard input, as shown in the following sample. In this version, the concurrent functions get a chance to complete while waiting for keyboard input:

```
func main() {
    go count(10, 30, 10)
    go count(40, 60, 10)
    go count(70, 120, 20)
    fmt.Scanln() // blocks for kb input
}
```

golang.fyi/ch09/goroutine1.go

Goroutines may also be defined as function literals directly in the `go` statement, as shown in this updated version of the example shown in the following code snippet:

```
func main() {  
    go count(10, 30, 10)  
    go func() {  
        count(40, 60, 10)  
    }()  
    ...  
}
```

golang.fyi/ch09/goroutine2.go

The function literal provides a convenient idiom that allows programmers to assemble logic directly at the site of the `go` statement. When using the `go` statement with a function literal, it is treated as a regular closure with lexical access to non-local variables, as shown in the following example:

```
func main() {  
    start := 0  
    stop := 50  
    step := 5  
    go func() {  
        count(start, stop, step)  
    }()  
}
```

golang.fyi/ch09/goroutine3.go

In the previous code, the goroutine is able to access and use the variables `start`, `stop`, and `step`. This is safe as long as the variables captured in the closure are not expected to change after the goroutine starts. If these values are updated outside of the closure, it may create race conditions causing the goroutine to read unexpected values by the time it is scheduled to run.

The following snippet shows an example where the goroutine closure captures the variable `j` from the loop:

```
func main() {
    starts := []int{10, 40, 70, 100}
    for _, j := range starts{
        go func() {
            count(j, j+20, 10)
        }()
    }
}
```

golang.fyi/ch09/goroutine4.go

Since `j` is updated with each iteration, it is impossible to determine what value will be read by the closure. In most cases, the goroutine closures will see the last updated value of `j` by the time they are executed. This can be easily fixed by passing the variable as a parameter in the function literal for the goroutine, as shown here:

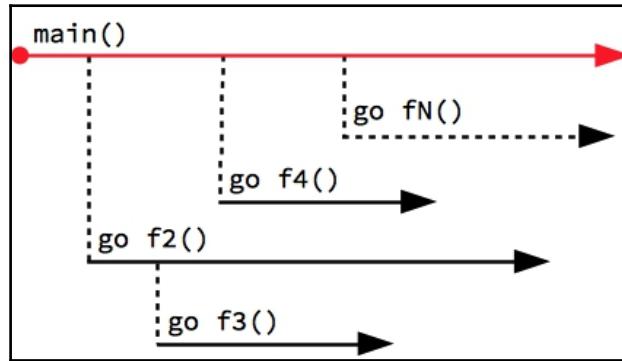
```
func main() {
    starts := []int{10, 40, 70, 100}
    for _, j := range starts{
        go func(s int) {
            count(s, s+20, 10)
        }(j)
    }
}
```

golang.fyi/ch09/goroutine5.go

The goroutine closures, invoked with each loop iteration, receive a copy of the `j` variable via the function parameter. This creates a local copy of the `j` value with the proper value to be used when the goroutine is scheduled to run.

Goroutine scheduling

In general, all goroutines run independently of each other, as depicted in the following illustration. A function that creates a goroutine does not wait for it to return, it continues with its own execution stream unless there is a blocking condition. Later, the chapter covers synchronization idioms to coordinate goroutines:



Go's runtime scheduler uses a form of cooperative scheduling to schedule goroutines. By default, the scheduler will allow a running goroutine to execute to completion. However, the scheduler will automatically yield to another goroutine for execution if one of the following events occurs:

- A `go` statement is encountered in the executing goroutine
- A channel operation is encountered (channels are covered later)
- A blocking system call (file or network IO for instance) is encountered
- After the completion of a garbage collection cycle

The scheduler will schedule a queued goroutines ready to enter execution when one of the previous events is encountered in a running goroutine. It is important to point out that the scheduler makes no guarantee of the order of execution of goroutines. When the following code snippet is executed, for instance, the output will be printed in an arbitrary order for each run:

```
func main() {
    go count(10, 30, 10)
    go count(40, 60, 10)
    go count(70, 120, 20)
    fmt.Scanln() // blocks for kb input
}
func count(start, stop, delta int) {
    for i := start; i <= stop; i += delta {
        fmt.Println(i)
    }
}
```

golang.fyi/ch09/goroutine1.go

The following shows possible output for the previous program:

```
10
70
90
110
40
50
60
20
30
```

Channels

When talking about concurrency, one of the natural concerns that arises is that of data safety and synchronization among concurrently executing code. If you have done concurrent programming in languages such as Java or C/C++, you are likely familiar with the, sometimes brittle, choreography required to ensure running threads can safely access shared memory values to achieve communication and synchronization between threads.

This is one area where Go diverges from its C lineage. Instead of having concurrent code communicate by using shared memory locations, Go uses channels as a conduit between running goroutines to communicate and share data. The blog post *Effective Go* (https://golang.org/doc/effective_go.html) has reduced this concept to the following slogan:

Do not communicate by sharing memory; instead, share memory by communicating.



The concept of channel has its roots in **communicating sequential processes (CSP)**, work done by renowned computer scientist C. A. Hoare, to model concurrency using communication primitives. As will be discussed in this section, channels provide the means to synchronize and safely communicate data between running goroutines.

This section discusses the Go channel type and provides insights into its characteristics. Later, you will learn how to use channels to craft concurrent programs.

The Channel type

The channel type declares a conduit within which only values of a given element type may be sent or received by the channel. The `chan` keyword is used to specify a channel type, as shown in the following declaration format:

`chan <element type>`

The following code snippet declares a bidirectional channel type, `chan int`, assigned to the variable `ch`, to communicate integer values:

```
func main() {
    var ch chan int
    ...
}
```

Later in the chapter, we will learn how to use the channel to send data between concurrent portions of a running program.

The send and receive operations

Go uses the `<-` (arrow) operator to indicate data movement within a channel. The following table summarizes how to send or receive data from a channel:

Example	Operation	Description
<code>intCh <- 12</code>	Send	When the arrow is placed to the left of the value, variable or expression, it indicates a send operation to the channel it points to. In this example, 12 is sent into channel <code>intCh</code> .
<code>value := <- intCh</code>	Receive	When the <code><-</code> operator is place to the left of a channel, it indicates a receive operation from the channel. The <code>value</code> variable is assigned the value received from the <code>intCh</code> channel.

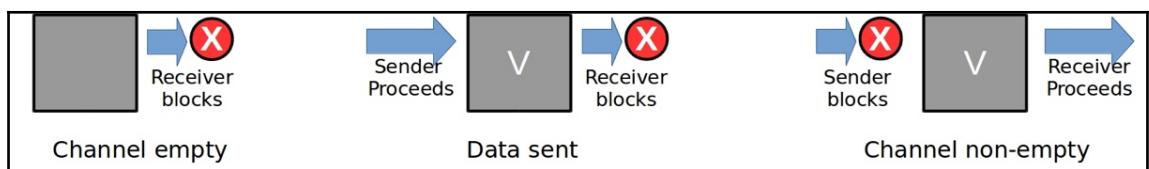
An uninitialized channel has a *nil* zero value and must be initialized using the built-in `make` function. As will be discussed in the following sections, a channel can be initialized as either unbuffered or buffered, depending on its specified capacity. Each of type of channel has different characteristics that are leveraged in different concurrency constructs.

Unbuffered channel

When the `make` function is invoked without the capacity argument, it returns a bidirectional *unbuffered* channel. The following snippet shows the creation of an unbuffered channel of type `chan int`:

```
func main() {
    ch := make(chan int) // unbuffered channel
    ...
}
```

The characteristics of an unbuffered channel are illustrated in the following figure:



The sequence in the preceding figure (from left to right) shows how the unbuffered channel works:

- If the channel is empty, the receiver blocks until there is data
- The sender can send only to an empty channel and blocks until the next receive operation
- When the channel has data, the receiver can proceed to receive the data.

Sending to an unbuffered channel can easily cause a *deadlock* if the operation is not wrapped in a goroutine. The following code will block after sending 12 to the channel:

```
func main() {  
    ch := make(chan int)  
    ch <- 12 // blocks  
    fmt.Println(<-ch)  
}
```

golang.fyi/ch09/chan-unbuff0.go

When you run the previous program, you will get the following result:

```
$> go run chan-unbuff0.go  
fatal error: all goroutines are asleep - deadlock!
```

Recall that the sender blocks immediately upon sending to an unbuffered channel. This means any subsequent statement, to receive from the channel for instance, becomes unreachable, causing a deadlock. The following code shows the proper way to send to an unbuffered channel:

```
func main() {  
    ch := make(chan int)  
    go func() { ch <- 12 }()  
    fmt.Println(<-ch)  
}
```

golang.fyi/ch09/chan-unbuff1.go

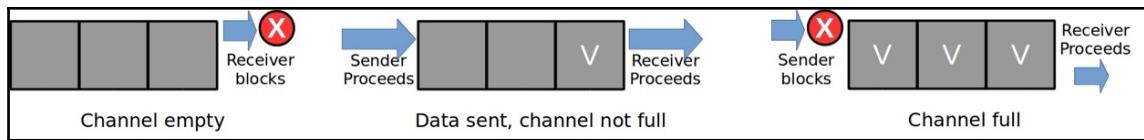
Notice that the send operation is wrapped in an anonymous function invoked as a separate goroutine. This allows the `main` function to reach the receive operation without blocking. As you will see later, this blocking property of unbuffered channels is used extensively as a synchronization and coordination idioms between goroutines.

Buffered channel

When the `make` function uses the `capacity` argument, it returns a bidirectional *buffered* channel, as shown in the following snippet:

```
func main()
    ch := make(chan int, 3) // buffered channel
}
```

The previous code will create a buffered channel with a capacity of 3. The buffered channel operates as a first-in-first-out blocking queue, as illustrated in the following figure:



The buffered channel depicted in the preceding figure has the following characteristics:

- When the channel is empty, the receiver blocks until there is at least one element
- The sender always succeeds as long as the channel is not at capacity
- When the channel is at capacity, the sender blocks until at least one element is received

Using a buffered channel, it is possible to send and receive values within the same goroutine without causing a deadlock. The following shows an example of sending and receiving using a buffered channel with a capacity of 4 elements:

```
func main() {
    ch := make(chan int, 4)
    ch <- 2
    ch <- 4
    ch <- 6
    ch <- 8

    fmt.Println(<-ch)
    fmt.Println(<-ch)
    fmt.Println(<-ch)
```

```
    fmt.Println(<-ch)
}

golang.fyi/ch09/chan0.go
```

The code in the previous example is able to send the values 2, 4, 6, and 8 to the `ch` channel without the risk of blocking. The four `fmt.Println(<-ch)` statements are used to receive the values buffered in the channel successively. However, if a fifth send operation is added, prior to the first receive, the code will deadlock as highlighted in the following snippet:

```
func main() {
    ch := make(chan int, 4)
    ch <- 2
    ch <- 4
    ch <- 6
    ch <- 8
    ch <- 10
    fmt.Println(<-ch)
    ...
}
```

Later in the chapter, you will read more about idiomatic and safe ways to use channels for communications.

Unidirectional channels

At declaration, a channel type may also include a unidirectional operator (using the `<-` arrow again) to indicate whether a channel is send-only or receive-only, as listed in the following table:

Declaration	Operation
<code><- chan <element type></code>	Declares a receive-only channel as shown later. <code>var Ch <-chan int</code>
<code>chan <-<element type></code>	Declares a send-only channel as shown later. <code>var Ch chan <-int</code>

The following code snippet shows function `makeEvenNums` with a send-only channel argument of type `chan <- int`:

```
func main() {
    ch := make(chan int, 10)
    makeEvenNums(4, ch)
```

```
    fmt.Println(<-ch)
    fmt.Println(<-ch)
    fmt.Println(<-ch)
    fmt.Println(<-ch)
}

func makeEvenNums(count int, in chan<- int) {
    for i := 0; i < count; i++ {
        in <- 2 * i
    }
}
```

golang.fyi/ch09/chan1.go

Since the directionality of the channel is baked in the type, access violations will be detected at compile time. So in the previous example, the `in` channel can only be used for receive operations.

A bidirectional channel can be converted to a unidirectional channel explicitly or automatically. For instance, when `makeEvenNums()` is called from `main()`, it receives the bidirectional channel `ch` as a parameter. The compiler automatically converts the channel to the appropriate type.

Channel length and capacity

The `len` and `cap` functions can be used to return a channel's length and capacity respectively. The `len` function returns the current number of elements queued in the channel prior to being read by a receiver. For instance, the following code snippet will print **2**:

```
func main() {
    ch := make(chan int, 4)
    ch <- 2
    ch <- 2
    fmt.Println(len(ch))
}
```

The `cap` function returns the declared capacity of the channel type which, unlike length, remains constant throughout the life of the channel.



An unbuffered channel has a length and a capacity of zero.

Closing a channel

Once a channel is initialized it is ready for send and receive operations. A channel will remain in that open state until it is forcibly closed using the built-in *close* function, as shown in the following example:

```
func main() {  
    ch := make(chan int, 4)  
    ch <- 2  
    ch <- 4  
    close(ch)  
    // ch <- 6 // panic, send on closed channel  
  
    fmt.Println(<-ch)  
    fmt.Println(<-ch)  
    fmt.Println(<-ch) // closed, returns zero value for element  
  
}
```

golang.fyi/ch09/chan2.go

Once a channel is closed, it has the following properties:

- Subsequent send operations will cause a program to panic
- Receive operations never block (regardless of whether buffered or unbuffered)
- All receive operations return the zero value of the channel's element type

In the previous snippet, the `ch` channel is closed after two send operations. As indicated in the comment, a third send operation would cause a panic because the channel is closed. On the receiving side, the code gets the two elements in the channel before it is closed. A third receive operation returns 0, the zero value for the channel's elements.

Go offers a long form of the receive operation that returns the value read from the channel followed by a Boolean indicating the closed status of the channel. This can be used to properly handle the zero value from a closed channel, as shown in the following example:

```
func main() {
    ch := make(chan int, 4)
    ch <- 2
    ch <- 4
    close(ch)

    for i := 0; i < 4; i++ {
        if val, opened := <-ch; opened {
            fmt.Println(val)
        } else {
            fmt.Println("Channel closed!")
        }
    }
}
```

golang.fyi/ch09/chan3.go

Writing concurrent programs

Up to this point, the discussions about goroutines and channels remained deliberately separated to ensure that each topic is properly covered. However, the true power of channels and goroutines are realized when they are combined to create concurrent programs, as covered in this section.

Synchronization

One of the primary uses of channels is synchronization between running goroutines. To illustrate this use case, let us examine the following code, which implements a word histogram. The program reads the words from the `data` slice then, on a separate goroutine, collects the occurrence of each word:

```
func main() {
    data := []string{
        "The yellow fish swims slowly in the water",
        "The brown dog barks loudly after a drink ...",
        "The dark bird bird of prey lands on a small ...",
    }

    histogram := make(map[string]int)
```

```
done := make(chan bool)

// splits and count words
go func() {
    for _, line := range data {
        words := strings.Split(line, " ")
        for _, word := range words {
            word = strings.ToLower(word)
            histogram[word]++
        }
    }
    done <- true
}()

if <-done {
    for k, v := range histogram {
        fmt.Printf("%s\t%d\n", k, v)
    }
}
}
```

golang.fyi/ch09/pattern0.go

The code in the previous example uses `done := make(chan bool)` to create the channel that will be used to synchronize the two running goroutines in the program. The `main` function launches a secondary goroutine, which does the word counting, and then it continues execution until it blocks at the `<-done` expression, causing it to wait.

In the meantime, the secondary goroutine runs until it completes its loop. Then, it sends a value to the `done` channel with `done <- true`, causing the blocked `main` routine to become unblocked and continues with its execution.



The previous code has a bug that may cause a race condition. A correction will be introduced later in the chapter.

In the previous example, the code allocates and actually sends a Boolean value that is used for the synchronization. Upon further inspection, it is clear that the value in the channel is irrelevant and we simply want it to signal. So, we can further distill the synchronization idiom into a colloquial form that is presented in the following code snippet:

```
func main() {
    ...
    histogram := make(map[string]int)
    done := make(chan struct{})
```

```
// splits and count
go func() {
    defer close(done) // closes channel upon fn return
    for _, line := range data {
        words := strings.Split(line, " ")
        for _, word := range words {
            word = strings.ToLower(word)
            histogram[word]++
        }
    }
}()

<-done // blocks until closed

for k, v := range histogram {
    fmt.Printf("%s\t%d\n", k, v)
}
}
```

golang.fyi/ch09/pattern1.go

This version of the code achieves goroutine synchronization using:

- The done channel, declared as type `chan struct{}`
- The main goroutine blocks at receive expression `<-done`
- When the done channel is closed, all receivers succeed without blocking

Although the signaling is done using different constructs, this version of the code is equivalent to the first version (`pattern0.go`). The empty `struct{}` type stores no value and it is used strictly for signaling. This version of the code closes the done channel (instead of sending a value). This has the effect of allowing the main goroutine to unblock and continue execution.

Streaming data

A natural use of channels is to stream data from one goroutine to another. This pattern is quite common in Go code and for it to work, the followings must be done:

- Continuously send data on a channel
- Continuously receive the incoming data from that channel
- Signal the end of the stream so the receiver may stop

As you will see, all of this can be done using a single channel. The following code snippet is a rewrite of the previous example. It shows how to use a single channel to stream data from one goroutine to another. The same channel is also used as a signaling device to indicate the end of the stream:

```
func main() {
    ...
    histogram := make(map[string]int)
    wordsCh := make(chan string)

    // splits lines and sends words to channel
    go func() {
        defer close(wordsCh) // close channel when done
        for _, line := range data {
            words := strings.Split(line, " ")
            for _, word := range words {
                word = strings.ToLower(word)
                wordsCh <- word
            }
        }
    }()
}

// process word stream and count words
// loop until wordsCh is closed
for {
    word, opened := <-wordsCh
    if !opened {
        break
    }
    histogram[word]++
}

for k, v := range histogram {
    fmt.Printf("%s\t(%d)\n", k, v)
}
}
```

golang.fyi/ch09/pattern2.go

This version of the code produces the word histogram as before, but introduces a different approach. This is accomplished using the highlighted portion of the code shown in the following table:

Code	Description
<code>wordsCh := make(chan string)</code>	The channel used to stream data.
<code>wordsCh <- word</code>	The sender goroutine loops through the text line and sends a word at a time. It then blocks until the word is received by the receiving (main) goroutine.
<code>defer close(wordsCh)</code>	As the words are continuously received (see later), the sender goroutine closes the channel when it is done. This will be the signal to the receiver that it should also stop.
<code>for { word, opened := <- wordsCh if !opened { break } histogram[word]++ }</code>	This is the receiver code. It is placed in a loop since it does not know ahead of time how much data to expect. With each iteration of the loop, the code does the following: <ul style="list-style-type: none">• Pulls the data from the channel• Checks the open status of the channel• If closed, break out of the loop• Otherwise record histogram

Using for...range to receive data

The previous pattern is so common in Go that the idiom is built into the language in the form of the following for...range statement:

`for <element> := range <channel>{...}`

With each iteration, this for...range statement will block until it receives incoming data from the indicated channel, as shown in the following snippet:

```
func main() {  
    ...  
    go func() {  
        defer close(wordsCh)  
        for _, line := range data {  
            words := strings.Split(line, " ")  
            for _, word := range words {  
                word = strings.ToLower(word)  
                wordsCh <- word  
            }  
        }  
    }  
}
```

```
        }
    }()

for word := range wordsCh {
    histogram[word]++
}
...
}
```

golang.fyi/ch09/pattern3.go

The previous code shows the an updated version of the code using a for-range statement, `for word := range wordsCh`. It successively emits the received value from the `wordsCh` channel. When the channel is closed (from the goroutine), the loop automatically breaks.



Always remember to close the channel so receivers are signaled properly. Otherwise, the program may enter into a deadlock which could cause a panic.

Generator functions

Channels and goroutines provide a natural substrate for implementing a form of producer/consumer pattern using generator functions. In this approach, a goroutine is wrapped in a function which generates values that are sent via a channel returned by the function. The consumer goroutine receives these values as they are generated.

The word histogram has been updated to use this pattern, as shown in the following code snippet:

```
func main() {
    data := []string{"The yellow fish swims...", ...}
    histogram := make(map[string]int)

    words := words(data) // returns handle to data channel
    for word := range words {
        histogram[word]++
    }
    ...

    // generator function that produces data
    func words(data []string) <-chan string {
        out := make(chan string)
        go func() {

```

```
    defer close(out) // closes channel upon fn return
    for _, line := range data {
        words := strings.Split(line, " ")
        for _, word := range words {
            word = strings.ToLower(word)
            out <- word
        }
    }
}()
```

```
return out
}
```

golang.fyi/ch09/pattern4.go

In this example, the generator function, declared as `func words(data []string) <- chan string`, returns a receive-only channel of string elements. The consumer function, in this instance `main()`, receives the data emitted by the generator function, which is processed using a `for...range` loop.

Selecting from multiple channels

Sometimes it is necessary for concurrent programs to handle send and receive operations for multiple channels at the same time. To facilitate such endeavor, the Go language supports the `select` statement that multiplexes selection among multiple send and receive operations:

```
select {
    case <send_or_receive_expression>:
    default:
}
```

The `case` statement operates similarly to a `switch` statement with `case` clauses. The `select` statement, however, selects one of the send or receive cases which succeeded. If two or more communication cases happen to be ready at the same time, one will be selected at random. The `default` case is always selected when no other cases succeed.

The following snippet updates the histogram code to illustrate the use of the `select` statement. The generator function `words` select between two channels, `out` to send data as before and a new channel `stopCh`, passed as a parameter, which is used to detect an interruption signal to stop sending data:

```
func main() {
    ...
    histogram := make(map[string]int)
    stopCh := make(chan struct{}) // used to signal stop

    words := words(stopCh, data) // returns handle to channel
    for word := range words {
        if histogram["the"] == 3 {
            close(stopCh)
        }
        histogram[word]++
    }
    ...
}

func words(stopCh chan struct{}, data []string) <-chan string {
    out := make(chan string)
    go func() {
        defer close(out) // closes channel upon fn return
        for _, line := range data {
            words := strings.Split(line, " ")
            for _, word := range words {
                word = strings.ToLower(word)
                select {
                    case out <- word:
                    case <-stopCh: // succeeds first when close
                        return
                }
            }
        }
    }()
    return out
}
```

golang.fyi/ch09/pattern5.go

In the previous code snippet, the `words` generator function will select the first communication operation that succeeds: `out <- word` or `<-stopCh`. As long as the consumer code in `main()` continues to receive from the `out` channel, the send operation will succeed first. Notice, however, the code in `main()` closes the `stopCh` channel when it encounters the third instance of "the". When that happens, it will cause the receive case, in the `select` statement, to proceed first causing the goroutine to return.

Channel timeout

One popular idiom that is commonly encountered with Go concurrency is the use of the `select` statement, introduced previously, to implement timeouts. This works by using the `select` statement to wait for a channel operation to succeed within a given time duration using the API from the `time` package (<https://golang.org/pkg/time/>).

The following code snippet shows a version of the word histogram example that times out if the program takes longer than 200 microseconds to count and print the words:

```
func main() {
    data := []string{...}
    histogram := make(map[string]int)
    done := make(chan struct{})

    go func() {
        defer close(done)
        words := words(data) // returns handle to channel
        for word := range words {
            histogram[word]++
        }
        for k, v := range histogram {
            fmt.Printf("%s\t%d\n", k, v)
        }
    }()
}

select {
case <-done:
    fmt.Println("Done counting words!!!!")
case <-time.After(200 * time.Microsecond):
    fmt.Println("Sorry, took too long to count.")
}
func words(data []string) <-chan string {...}
```

golang.fyi/ch09/pattern6.go

This version of the histogram example introduces the `done` channel, which is used to signal when processing is done. In the `select` statement, the receive operation `case <- done :` blocks until the goroutine closes the `done` channel. Also in the `select` statement, the `time.After()` function returns a channel which will close after the indicated duration. If the 200 microseconds elapse before `done` is closed, the channel from `time.After()` will close first, causing the timeout case to succeed first.

The sync package

There are instances when accessing shared values using traditional methods are simpler and more appropriate than the use of channels. The `sync` package (<https://golang.org/pkg/sync/>) provides several synchronization primitives including mutual exclusion (mutex) locks and synchronization barriers for safe access to shared values, as discussed in this section.

Synchronizing with mutex locks

Mutex locks allow serial access of shared resources by causing goroutines to block and wait until locks are released. The following sample illustrates a typical code scenario with the `Service` type, which must be started before it is ready to be used. After the service has started, the code updates an internal bool variable, `started`, to store its current state:

```
type Service struct {
    started bool
    stpCh   chan struct{}
    mutex   sync.Mutex
}
func (s *Service) Start() {
    s.stpCh = make(chan struct{})
    go func() {
        s.mutex.Lock()
        s.started = true
        s.mutex.Unlock()
        <-s.stpCh // wait to be closed.
    }()
}
func (s *Service) Stop() {
    s.mutex.Lock()
    defer s.mutex.Unlock()
    if s.started {
        s.started = false
        close(s.stpCh)
```

```
    }
}

func main() {
    s := &Service{}
    s.Start()
    time.Sleep(time.Second) // do some work
    s.Stop()
}
```

golang.fyi/ch09/sync2.go

The previous code snippet uses variable `mutex`, of type `sync.Mutex`, to synchronize access to the shared variable `started`. For this to work effectively, all contentious areas where the `started` variable is updated must use the same lock with successive calls to `mutex.Lock()` and `mutex.Unlock()`, as shown in the code.

One idiom you will often encounter is to embed the `sync.Mutex` type directly inside a struct, as shown in the next code snippet. This has the effect of promoting the `Lock()` and `Unlock()` methods as part of the struct itself:

```
type Service struct {
    ...
    sync.Mutex
}

func (s *Service) Start() {
    s.stpCh = make(chan struct{})
    go func() {
        s.Lock()
        s.started = true
        s.Unlock()
        <-s.stpCh // wait to be closed.
    }()
}

func (s *Service) Stop() {
    s.Lock()
    defer s.Unlock()
    ...
}
```

golang.fyi/ch09/sync3.go

The `sync` package also offers the `RWMutex` (read-write mutex), which can be used in cases where there is one writer that updates the shared resource, while there may be multiple readers. The writer would update the resource using a full lock, as before. However, readers use the `RLock()`/`RUnlock()` method pair (for read-lock/read-unlock respectively) to apply a read-only lock when reading the shared resource. The `RWMutex` type is used in the next section, *Synchronizing Access to Composite Values*.

Synchronizing access to composite values

The previous section discussed concurrency safety when sharing access to simple values. The same level of care must be applied when sharing access to composite type values such as maps and slices, since Go does not offer concurrency-safe version of these types, as illustrated in the following example:

```
type Service struct {
    started bool
    stpCh   chan struct{}
    mutex   sync.RWMutex
    cache   map[int]string
}

func (s *Service) Start() {
    ...
    go func() {
        s.mutex.Lock()
        s.started = true
        s.cache[1] = "Hello World"
        ...
        s.mutex.Unlock()
        <-s.stpCh // wait to be closed.
    }()
}
...
func (s *Service) Serve(id int) {
    s.mutex.RLock()
    msg := s.cache[id]
    s.mutex.RUnlock()
    if msg != "" {
        fmt.Println(msg)
    } else {
        fmt.Println("Hello, goodbye!")
    }
}
```

golang.fyi/ch09/sync4.go

The preceding code uses a `sync.RWMutex` variable (see preceding section, *Synchronizing with Mutex Locks*) to manage the locks when accessing the map variable `cache`. The code wraps the update operation to the `cache` variable within a pair of method calls, `mutex.Lock()` and `mutex.Unlock()`. However, when reading values from the `cache` variable, the `mutex.RLock()` and `mutex.RUnlock()` methods are used to provide concurrency safety.

Concurrency barriers with sync.WaitGroup

Sometimes when working with goroutines, you may need to create a synchronization barrier where you wish to wait for all running goroutines to finish before proceeding. The `sync.WaitGroup` type is designed for such a scenario, allowing multiple goroutines to rendezvous at specific point in the code. Using `WaitGroup` requires three things:

- The number of participants in the group via the `Add` method
- Each goroutine calls the `Done` method to signal completion
- Use the `Wait` method to block until all goroutines are done

`WaitGroup` is often used as a way to implement work distribution patterns. The following code snippet illustrates work distribution to calculate the sum of multiples of 3 and 5 up to `MAX`. The code uses the `WaitGroup` variable, `wg`, to create a concurrency barrier that waits for two goroutines to calculate the partial sums of the numbers, then gathers the result after all goroutines are done:

```
const MAX = 1000

func main() {
    values := make(chan int, MAX)
    result := make(chan int, 2)
    var wg sync.WaitGroup
    wg.Add(2)
    go func() { // gen multiple of 3 & 5 values
        for i := 1; i < MAX; i++ {
            if (i%3) == 0 || (i%5) == 0 {
                values <- i // push downstream
            }
        }
        close(values)
    }()
    work := func() { // work unit, calc partial result
        defer wg.Done()
        r := 0
```

```
        for i := range values {
            r += i
        }
        result <- r
    }

    // distribute work to two goroutines
    go work()
    go work()

    wg.Wait()                      // wait for both goroutines
    total := <-result + <-result // gather partial results
    fmt.Println("Total:", total)
}
```

golang.fyi/ch09/sync5.go

In the previous code, the method call, `wg.Add(2)`, configures the `WaitGroup` variable `wg` because the work is distributed between two goroutines. The `work` function calls `defer wg.Done()` to decrement the `WaitGroup` counter by one every time it is completed.

Lastly, the `wg.Wait()` method call blocks until its internal counter reaches zero. As explained previously, this will happen when both goroutines' `work` running function complete successfully. When that happens, the program unblocks and gathers the partial results. It is important to remember that `wg.Wait()` will block indefinitely if its internal counter never reaches zero.

Detecting race conditions

Debugging concurrent code with a race condition can be time consuming and frustrating. When a race condition occurs, it is usually inconsistent and displays little to no discernible pattern. Fortunately, since Version 1.1, Go has included a race detector as part of its command-line tool chain. When building, testing, installing, or running Go source code, simply add the `-race` command flag to enable the race detector instrumentation of your code.

For instance, when the source file `golang.fyi/ch09/sync1.go` (a code with a race condition) is executed with the `-race` flag, the compiler's output shows the offending goroutine locations that caused the race condition, as shown in the following output:

```
$> go run -race sync1.go
=====
WARNING: DATA RACE
Read by main goroutine:
    main.main()
/github.com/vladimirvivien/learning-go/ch09/sync1.go:28 +0x8c

Previous write by goroutine 5:
    main.(*Service).Start.func1()
/github.com/vladimirvivien/learning-go/ch09/sync1.go:13 +0x2e

Goroutine 5 (running) created at:
    main.(*Service).Start()
/github.com/vladimirvivien/learning-go/ch09/sync1.go:15 +0x99
    main.main()
/github.com/vladimirvivien/learning-go/ch09/sync1.go:26 +0x6c
=====
Found 1 data race(s)
exit status 66
```

The race detector lists the line numbers where there is concurrent access to shared values. It lists the *read* operations followed by the locations where *write* operations may happen concurrently. Racy conditions in code can go unnoticed, even in well-tested code, until it manifests itself randomly. If you are writing concurrent code, it is highly recommended that you integrate the race detector as part of your testing suite.

Parallelism in Go

So far, the discussion in this chapter has focused on synchronizing concurrent programs. As was mentioned earlier in the chapter, the Go runtime scheduler automatically multiplexes and schedules goroutines across available OS-managed threads. This means concurrent programs that can be parallelized have the ability to take advantage of the underlying processor cores with little to no configuration. For instance, the following code cleanly segregates its work unit (to calculate sums of multiples of 3 and 5) to be calculated by launching `workers` number of goroutines:

```
const MAX = 1000
const workers = 2

func main() {
```

```
values := make(chan int)
result := make(chan int, workers)
var wg sync.WaitGroup

go func() { // gen multiple of 3 & 5 values
    for i := 1; i < MAX; i++ {
        if (i%3) == 0 || (i%5) == 0 {
            values <- i // push downstream
        }
    }
    close(values)
}()

work := func() { // work unit, calc partial result
    defer wg.Done()
    r := 0
    for i := range values {
        r += i
    }
    result <- r
}

//launch workers
wg.Add(workers)
for i := 0; i < workers; i++ {
    go work()
}

wg.Wait() // wait for all groutines
close(result)
total := 0
// gather partial results
for pr := range result {
    total += pr
}
fmt.Println("Total:", total)
}
```

golang.fyi/ch09/sync6.go

The previous code will automatically launch each goroutine, with `go work()`, in parallel when executed on a multi-core machine. The Go runtime scheduler, by default, will create a number of OS-backed threads for scheduling that is equal to the number of CPU cores. That quantity is identified by runtime value called `GOMAXPROCS`.

The GOMAXPROCS value can be explicitly changed to influence the number threads that are made available to the scheduler. That value can be changed using a command-line environment variable with the same name. GOMAXPROCS can also be updated in the using function `GOMAXPROCS()` from the *runtime* package (<https://golang.org/pkg/runtime>). Either approach allows programmers to fine-tune the number of threads that will participate in scheduling goroutines.

Summary

Concurrency can be a complex topic in any language. This chapter covered the major topics to guide readers around the use of concurrency primitives in the Go language. The first section of the chapter outlined the crucial properties of goroutines, including the creation and usage of the `go` statement. Next, the chapter covered the mechanism of Go's runtime scheduler and the notion of channels used for communication between running goroutines. Lastly, users were introduced to several concurrency patterns used to create concurrent programs using goroutines, channels, and the synchronization primitives from the `sync` package.

Next, you will be introduced to the standard APIs to do data input and output in Go.

10

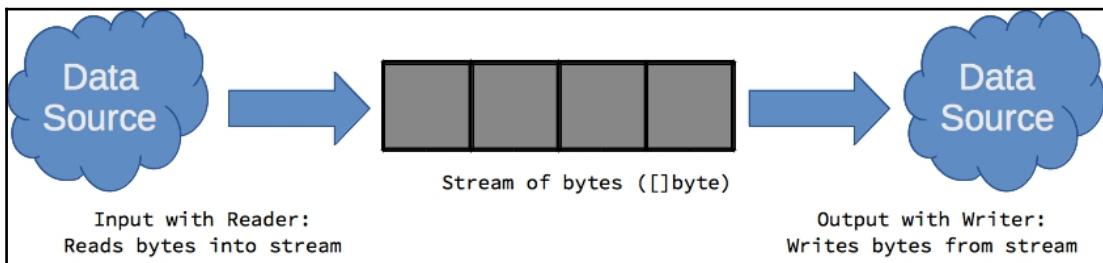
Data IO in Go

Previous chapters of this book focused mainly on fundamentals. In this and future chapters, readers are introduced to some of the powerful APIs provided by Go's standard library. This chapter discusses in detail how to input, process, transform, and output data using APIs from the standard library and their respective packages with the following topics:

- IO with readers and writers
- The io.Reader interface
- The io.Writer interface
- Working with the io package
- Working with files
- Formatted IO with fmt
- Buffered IO
- In-memory IO
- Encoding and decoding data

IO with readers and writers

Similar to other languages, such as Java, Go models data input and output as a stream that flows from sources to targets. Data resources, such as files, networked connections, or even some in-memory objects, can be modeled as streams of bytes from which data can be *read* or *written* to, as illustrated in the following figure:



The stream of data is represented as a **slice of bytes ([]byte)** that can be accessed for reading or writing. As we will explore in this chapter, the `io` package makes available the `io.Reader` interface to implement code that *reads* and transfers data from a source into a stream of bytes. Conversely, the `io.Writer` interface lets implementers create code that reads data from a provided stream of bytes and *writes* it as output to a target resource. Both interfaces are used extensively in Go as a standard idiom to express IO operations. This makes it possible to interchange readers and writers of different implementations and contexts with predictable results.

The `io.Reader` interface

The `io.Reader` interface, as shown in the following listing, is simple. It consists of a single method, `Read([]byte) (int, error)`, intended to let programmers implement code that *reads* data, from an arbitrary source, and transfers it into the provided slice of bytes.

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

The `Read` method returns the total number of bytes transferred into the provided slice and an error value (if necessary). As a guideline, implementations of the `io.Reader` should return an error value of `io.EOF` when the reader has no more data to transfer into stream `p`. The following shows the type `alphaReader`, a trivial implementation of the `io.Reader` that filters out non-alpha characters from its string source:

```
type alphaReader string

func (a alphaReader) Read(p []byte) (int, error) {
    count := 0
    for i := 0; i < len(a); i++ {
        if (a[i] >= 'A' && a[i] <= 'Z') ||
            (a[i] >= 'a' && a[i] <= 'z') {
            p[i] = a[i]
        }
        count++
    }
    return count, io.EOF
}

func main() {
    str := alphaReader("Hello! Where is the sun?")
    io.Copy(os.Stdout, &str)
    fmt.Println()
}
```

golang.fyi/ch10/reader0.go

Since values of the `alphaReader` type implement the `io.Reader` interface, they can participate anywhere a reader is expected as shown in the call to `io.Copy(os.Stdout, &str)`. This copies the stream of bytes emitted by the `alphaReader` variable into a writer interface, `os.Stdout` (covered later).

Chaining readers

Chances are the standard library already has a reader that you can reuse - so it is common to wrap an existing reader and use its stream as the source for the new implementation. The following snippet shows an updated version of `alphaReader`. This time, it takes an `io.Reader` as its source as shown in the following code:

```
type alphaReader struct {
    src io.Reader
}
```

```
func NewAlphaReader(source io.Reader) *alphaReader {
    return &alphaReader{source}
}

func (a *alphaReader) Read(p []byte) (int, error) {
    if len(p) == 0 {
        return 0, nil
    }
    count, err := a.src.Read(p) // p has now source data
    if err != nil {
        return count, err
    }
    for i := 0; i < len(p); i++ {
        if (p[i] >= 'A' && p[i] <= 'Z') ||
            (p[i] >= 'a' && p[i] <= 'z') {
            continue
        } else {
            p[i] = 0
        }
    }
    return count, io.EOF
}

func main() {
    str := strings.NewReader("Hello! Where is the sun?")
    alpha := NewAlphaReader(str)
    io.Copy(os.Stdout, alpha)
    fmt.Println()
}
```

golang.fyi/ch10/reader1.go

The main change to note in this version of the code is that the `alphaReader` type is now a struct which embeds an `io.Reader` value. When `alphaReader.Read()` is invoked, it calls the wrapped reader as `a.src.Read(p)`, which will inject the source data into byte slice `p`. Then the method loops through `p` and applies the filter to the data. Now, to use the `alphaReader`, it must first be provided with an existing reader which is facilitated by the `NewAlphaReader()` constructor function.

The advantages of this approach may not be obvious at first. However, by using an `io.Reader` as the underlying data source the `alphaReader` type is capable of reading from any reader implementation. For instance, the following code snippet shows how the `alphaReader` type can now be combined with an `os.File` to filter out non-alphabetic characters from a file (the Go source code itself):

```
...
func main() {
    file, _ := os.Open("./reader2.go")
    alpha := NewAlphaReader(file)
    io.Copy(os.Stdout, alpha)
    fmt.Println()
}
```

golang.fyi/ch10/reader2.go

The `io.Writer` interface

The `io.Writer` interface, as shown in the following code, is just as simple as its reader counterpart:

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

The interface requires the implementation of a single method, `Write(p []byte)` (`n int, e error`), that copies data from the provided stream `p` and *writes* that data to a sink resource such as an in-memory structure, standard output, a file, a network connection, or any number of `io.Writer` implementations that come with the Go standard library. The `Write` method returns the number of bytes copied from `p` followed by an `error` value if any was encountered.

The following code snippet shows the implementation of the `channelWriter` type, a writer that decomposes and serializes its stream that is sent over a Go channel as consecutive bytes:

```
type channelWriter struct {
    Channel chan byte
}

func NewChannelWriter() *channelWriter {
    return &channelWriter{
        Channel: make(chan byte, 1024),
```

```
    }

}

func (c *channelWriter) Write(p []byte) (int, error) {
    if len(p) == 0 {
        return 0, nil
    }

    go func() {
        defer close(c.Channel) // when done
        for _, b := range p {
            c.Channel <- b
        }
    }()
}

return len(p), nil
}
```

golang.fyi/ch10/writer1.go

The `Write` method uses a goroutine to copy each byte, from `p`, and sends it across the `c.Channel`. Upon completion, the goroutine closes the channel so that consumers are notified when to stop consuming from the channel. As an implementation convention, writers should not modify slice `p` or hang on to it. When an error occurs, the writer should return the current number of bytes processed and an error.

Using the `channelWriter` type is simple. You can invoke the `Write()` method directly or, as is more common, use `channelWriter` with other IO primitives in the API. For instance, the following snippet uses the `fmt.Fprint` function to serialize the "Stream me!" string as a sequence of bytes over a channel using `channelWriter`:

```
func main() {
    cw := NewChannelWriter()
    go func() {
        fmt.Fprint(cw, "Stream me!")
    }()

    for c := range cw.Channel {
        fmt.Printf("%c\n", c)
    }
}
```

golang.fyi/ch10/writer1.go

In the previous snippet, the serialized bytes, queued in the channel, are consumed using a `for...range` statement as they are successively printed. The following snippet shows another example where the content of a file is serialized over a channel using the same `channelWriter`. In this implementation, an `io.File` value and `io.Copy` function are used to source the data instead of the `fmt.Fprint` function:

```
func main() {
    cw := NewChannelWriter()
    file, err := os.Open("./writer2.go")
    if err != nil {
        fmt.Println("Error reading file:", err)
        os.Exit(1)
    }
    _, err = io.Copy(cw, file)
    if err != nil {
        fmt.Println("Error copying:", err)
        os.Exit(1)
    }

    // consume channel
    for c := range cw.Channel {
        fmt.Printf("%c\n", c)
    }
}
```

golang.fyi/ch10/writer2.go.

Working with the io package

The obvious place to start with IO is, well, the `io` package (<https://golang.org/pkg/io>). As we have already seen, the `io` package defines input and output primitives as the `io.Reader` and `io.Writer` interfaces. The following table summarizes additional functions and types, available in the `io` package, that facilitate streaming IO operations.

Function	Description
<code>io.Copy()</code>	<p>The <code>io.Copy</code> function (and its variants <code>io.CopyBuffer</code> and <code>io.CopyN</code>) make it easy to copy data from an arbitrary <code>io.Reader</code> source into an equally arbitrary <code>io.Writer</code> sink as shown in the following snippet:</p> <pre>data := strings.NewReader("Write me down.") file, _ := os.Create("./iocopy.data") io.Copy(file, data) golang.fyi/ch10/iocopy.go</pre>
<code>PipeReader</code> <code>PipeWriter</code>	<p>The <code>io</code> package includes the <code>PipeReader</code> and <code>PipeWriter</code> types that model IO operations as an in-memory pipe. Data is written to the pipe's <code>io.Writer</code> and can independently be read at the pipe's <code>io.Reader</code>. The following abbreviated snippet illustrates a simple pipe that writes a string to the writer <code>pw</code>. The data is then consumed with the <code>pr</code> reader and copied to a file:</p> <pre>file, _ := os.Create("./iopipe.data") pr, pw := io.Pipe() go func() { fmt.Fprint(pw, "Pipe streaming") pw.Close() }() wait := make(chan struct{}) go func() { io.Copy(file, pr) pr.Close() close(wait) }() <-wait //wait for pr to finish golang.fyi/ch10/iopipe.go</pre> <p>Note that the pipe writer will block until the reader completely consumes the pipe content or an error is encountered. Therefore, both the reader and writer should be wrapped in a goroutine to avoid deadlocks.</p>

io.TeeReader()	<p>Similar to the <code>io.Copy</code> function, <code>io.TeeReader</code> transfers content from a reader to a writer. However, the function also emits the copied bytes (unaltered) via a returned <code>io.Reader</code>. The TeeReader works well for composing multi-step IO stream processing. The following abbreviated snippet first calculates the SHA-1 hash of a file content using the TeeReader. The resulting reader, <code>data</code>, is then streamed to a gzip writer <code>zip</code>:</p> <pre>fin, _ := os.Open("./ioteeerdr.go") defer fin.Close() fout, _ := os.Create("./teereader.gz") defer fout.Close() zip := gzip.NewWriter(fout) defer zip.Close() sha := sha1.New() data := io.TeeReader(fin, sha) io.Copy(zip, data) fmt.Printf("SHA1 hash %x\n", sha.Sum(nil)) golang.fyi/ch10/ioteeerdr0.go</pre> <p>If we wanted to calculate both SHA-1 and MD5, we can update the code to nest the two TeeReader values as shown in the following snippet:</p> <pre>sha := sha1.New() md := md5.New() data := io.TeeReader(io.TeeReader(fin, md), sha,) io.Copy(zip, data) golang.fyi/ch10/ioteeerdr1.go</pre>
io.WriteString()	<p>The <code>io.WriteString</code> function writes the content of string into a specified writer. The following writes the content of a string to a file:</p> <pre>fout, err := os.Create("./iowritestr.data") if err != nil { fmt.Println(err) os.Exit(1) } defer fout.Close() io.WriteString(fout, "Hello there!\n") golang.fyi/ch10/iowritestr.go</pre>

io.LimitedReader	As its name suggests, the <code>io.LimitedReader</code> struct is a reader that reads only <code>N</code> number of bytes from the specified <code>io.Reader</code> . The following snippet will print the first 19 bytes from the string: <pre>str := strings.NewReader("The quick brown " + "fox jumps over the lazy dog") limited := &io.LimitedReader{R: str, N: 19} io.Copy(os.Stdout, limited) golang.fyi/ch10/iolimitedrdr.go \$> go run iolimitedrdr.go The quick brown fox</pre>
io.SectionReader	The <code>io.SectionReader</code> type implements seek and skip primitives by specifying an index (zero-based) where to start reading and an offset value indicating the number of bytes to read as shown in the following snippet: <pre>str := strings.NewReader("The quick brown" + "fox jumps over the lazy dog") section := io.NewSectionReader(str, 19, 23) io.Copy(os.Stdout, section) golang.fyi/ch10/iosectionrdr.go This example will print jumps over the lazy dog.</pre>
Package <code>io/ioutil</code>	The <code>io/ioutil</code> sub-package implements a small number of functions that provide utilitarian shortcuts to IO primitives such as file read, directory listing, temp directory creation, and file write.

Working with files

The `os` package (<https://golang.org/pkg/os/>) exposes the `os.File` type which represents a file handle on the system. The `os.File` type implements several IO primitives, including the `io.Reader` and `io.Writer` interfaces, which allows file content to be processed using the standard streaming IO API.

Creating and opening files

The `os.Create` function creates a new file with the specified path. If the file already exists, `os.Create` will overwrite it. The `os.Open` function, on the other hand, opens an existing file for reading.

The following source snippet opens an existing file and creates a copy of its content using the `io.Copy` function. One common, and recommended practice to notice is the deferred call to the method `Close` on the file. This ensures a graceful release of OS resources when the function exits:

```
func main() {
    f1, err := os.Open("./file0.go")
    if err != nil {
        fmt.Println("Unable to open file:", err)
        os.Exit(1)
    }
    defer f1.Close()

    f2, err := os.Create("./file0.bkp")
    if err != nil {
        fmt.Println("Unable to create file:", err)
        os.Exit(1)
    }
    defer f2.Close()

    n, err := io.Copy(f2, f1)
    if err != nil {
        fmt.Println("Failed to copy:", err)
        os.Exit(1)
    }

    fmt.Printf("Copied %d bytes from %s to %s\n",
        n, f1.Name(), f2.Name())
}
```

golang.fyi/ch10/file0.go

Function `os.OpenFile`

The `os.OpenFile` function provides generic low-level functionalities to create a new file or open an existing file with fine-grained control over the file's behavior and its permission. Nevertheless, the `os.Open` and `os.Create` functions are usually used instead as they provide a simpler abstraction than the `os.OpenFile` function.

The `os.OpenFile` function takes three parameters. The first one is the path of the file, the second parameter is a masked bit-field value to indicate the behavior of the operation (for example, read-only, read-write, truncate, and so on) and the last parameter is a POSIX-compliant permission value for the file.

The following abbreviated source snippet re-implements the file copy code, from earlier. This time, however, it uses the `os.FileOpen` function to demonstrate how it works:

```
func main() {
    f1, err := os.OpenFile("./file0.go", os.O_RDONLY, 0666)
    if err != nil {...}
    defer f1.Close()

    f2, err := os.OpenFile("./file0.bkp", os.O_WRONLY, 0666)
    if err != nil {...}
    defer f2.Close()

    n, err := io.Copy(f2, f1)
    if err != nil {...}

    fmt.Printf("Copied %d bytes from %s to %s\n",
        n, f1.Name(), f2.Name())
}
```

golang.fyi/ch10/file1.go



If you already have a reference to an OS file descriptor, you can also use the `os.NewFile` function to create a file handle in your program. The `os.NewFile` function is rarely used, as files are usually initialized using the file functions discussed previously.

Files writing and reading

We have already seen how to use the `os.Copy` function to move data into or out of a file. Sometimes, however, it will be necessary to have complete control over the logic that writes or reads file data. The following code snippet, for instance, uses the `WriteString` method from the `os.File` variable, `fout`, to create a text file:

```
func main() {
    rows := []string{
        "The quick brown fox",
        "jumps over the lazy dog",
    }
```

```
fout, err := os.Create("./fwrite.data")
if err != nil {
    fmt.Println(err)
    os.Exit(1)
}
defer fout.Close()

for _, row := range rows {
    fout.WriteString(row)
}
}
```

[golang.fyi/ch10/f fwrite0.go](https://golang.fyi/ch10/f	fwrite0.go)

If, however, the source of your data is not text, you can write raw bytes directly to the file as shown in the following source snippet:

```
func main() {
    data := [][]byte{
        []byte("The quick brown fox\n"),
        []byte("jumps over the lazy dog\n"),
    }
    fout, err := os.Create("./fwrite.data")
    if err != nil { ... }
    defer fout.Close()

    for _, out := range data {
        fout.Write(out)
    }
}
```

[golang.fyi/ch10/f fwrite0.go](https://golang.fyi/ch10/f	fwrite0.go)

As an `io.Reader`, reading from of the `io.File` type directly can be done using the `Read` method. This gives access to the content of the file as a raw stream of byte slices. The following code snippet reads the content of file `../ch0r/dict.txt` as raw bytes assigned to slice `p` up to 1024-byte chunks at a time:

```
func main() {
    fin, err := os.Open("../ch05/dict.txt")
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
    defer fin.Close()
    p := make([]byte, 1024)
    for {
```

```
    n, err := fin.Read(p)
    if err == io.EOF {
        break
    }
    fmt.Print(string(p[:n]))
}
}
```

golang.fyi/ch10/fileread.go

Standard input, output, and error

The `os` package includes three pre-declared variables, `os.Stdin`, `os.Stdout`, and `os.Stderr`, that represent file handles for standard input, output, and error of the OS respectively. The following snippet reads the file `f1` and writes its content to `io.Stdout`, standard output, using the `os.Copy` function (standard input is covered later):

```
func main() {
    f1, err := os.Open("./file0.go")
    if err != nil {
        fmt.Println("Unable to open file:", err)
        os.Exit(1)
    }
    defer f1.Close()

    n, err := io.Copy(os.Stdout, f1)
    if err != nil {
        fmt.Println("Failed to copy:", err)
        os.Exit(1)
    }

    fmt.Printf("Copied %d bytes from %s \n", n, f1.Name())
}
```

golang.fyi/ch10/osstd.go

Formatted IO with `fmt`

One of the most widely used packages for IO is `fmt` (<https://golang.org/pkg/fmt>). It comes with an amalgam of functions designed for formatted input and output. The most common usage of the `fmt` package is for writing to standard output and reading from standard input. This section also highlights other functions that make `fmt` a great tool for IO.

Printing to `io.Writer` interfaces

The `fmt` package offers several functions designed to write text data to arbitrary implementations of `io.Writer`. The `fmt.Fprint` and `fmt.Fprintln` functions write text with the default format while `fmt.Fprintf` supports format specifiers. The following code snippet writes a columnar formatted list of `metalloid` data to a specified text file using the `fmt.Fprintf` function:

```
type metalloid struct {
    name    string
    number int32
    weight float64
}

func main() {
    var metalloids = []metalloid{
        {"Boron", 5, 10.81},
        ...
        {"Polonium", 84, 209.0},
    }
    file, _ := os.Create("./metalloids.txt")
    defer file.Close()

    for _, m := range metalloids {
        fmt.Fprintf(
            file,
            "%-10s %-10d %-10.3f\n",
            m.name, m.number, m.weight,
        )
    }
}
```

golang.fyi/ch10/fmtfprint0.go

In the previous example, the `fmt.Fprintf` function uses format specifiers to write formatted text to the `io.File` `file` variable. The `fmt.Fprintf` function supports a large number of format specifiers whose proper treatment is beyond the scope of this text. Refer to the online documentation for complete coverage of these specifiers.

Printing to standard output

The `fmt.Print`, `fmt.Printf`, and `fmt.Println` have the exact same characteristics as the previous `Fprint`-series of functions seen earlier. Instead of an arbitrary `io.Writer` however, they write text to the standard output file handle `os.Stdout` (see the section *Standard output, input, and error* covered earlier).

The following abbreviated code snippet shows an updated version of the previous example that writes the list of metalloids to a standard output instead of a regular file. Note that it is the same code except for the use of the `fmt.Printf` instead of the `fmt.Fprintf` function:

```
type metalloid struct { ... }
func main() {
    var metalloids = []metalloid{
        {"Boron", 5, 10.81},
        ...
        {"Polonium", 84, 209.0},
    }

    for _, m := range metalloids {
        fmt.Printf(
            "%-10s %-10d %-10.3f\n",
            m.name, m.number, m.weight,
        )
    }
}
```

golang.fyi/ch10/fmtprint0.go

Reading from `io.Reader`

The `fmt` package also supports formatted reading of textual data from `io.Reader` interfaces. The `fmt.Fscanf` and `fmt.FscanfN` functions can be used to read multiple values, separated by spaces, into specified parameters. The `fmt.FscanfF` function supports format specifiers for a richer and flexible parsing of data input from `io.Reader` implementations.

The following abbreviated code snippet uses the function `fmt.Fscanf` for the formatted input of a space-delimited file (`planets.txt`) containing planetary data:

```
func main() {
    var name, hasRing string
    var diam, moons int

    // read data
    data, err := os.Open("./planets.txt")
    if err != nil {
        fmt.Println("Unable to open planet data:", err)
        return
    }
    defer data.Close()

    for {
        _, err := fmt.Fscanf(
            data,
            "%s %d %d %s\n",
            &name, &diam, &moons, &hasRing,
        )
        if err != nil {
            if err == io.EOF {
                break
            } else {
                fmt.Println("Scan error:", err)
                return
            }
        }
        fmt.Printf(
            "-10s -10d -6d -6s\n",
            name, diam, moons, hasRing,
        )
    }
}
```

golang.fyi/ch10/fmtfscan0.go

The code reads from the `io.File` variable `data`, until it encounters an `io.EOF` error indicating the end of the file. Each line of text it reads is parsed using format specifiers `"%s %d %d %s\n"` which matches the space-delimited layout of the records stored in the file. Each parsed token is then assigned to its respective variable `name`, `diam`, `moons`, and `hasRing`, which are printed to the standard output using `fm.Printf` function.

Reading from standard input

Instead of reading from an arbitrary `io.Reader`, the `fmt.Scan`, `fmt.Scanf`, and `fmt.Scanln` are used to read data from standard input file handle, `os.Stdin`. The following code snippet shows a simple program that reads text input from the console:

```
func main() {
    var choice int
    fmt.Println("A square is what?")
    fmt.Print("Enter 1=quadrilateral 2=rectagonal:")

    n, err := fmt.Scanf("%d", &choice)
    if n != 1 || err != nil {
        fmt.Println("Follow directions!")
        return
    }
    if choice == 1 {
        fmt.Println("You are correct!")
    } else {
        fmt.Println("Wrong, Google it.")
    }
}
```

golang.fyi/ch10/fmtscan1.go

In the previous program, the `fmt.Scanf` function parses the input using the format specifier `%d` to read an integer value from the standard input. The function will throw an error if the value read does not match exactly the specified format. For instance, the following shows what happens when character `D` is read instead of an integer:

```
$> go run fmtscan1.go
A square is what?
Enter 1=quadrilateral 2=rectagonal: D
Follow directions!
```

Buffered IO

Most IO operations covered so far have been unbuffered. This implies that each read and write operation could be negatively impacted by the latency of the underlying OS to handle IO requests. Buffered operations, on the other hand, reduces latency by buffering data in internal memory during IO operations. The `bufio` package (<https://golang.org/pkg/bufio/>) offers functions for buffered read and write IO operations.

Buffered writers and readers

The `bufio` package offers several functions to do buffered writing of IO streams using an `io.Writer` interface. The following snippet creates a text file and writes to it using buffered IO:

```
func main() {
    rows := []string{
        "The quick brown fox",
        "jumps over the lazy dog",
    }

    fout, err := os.Create("./filewrite.data")
    writer := bufio.NewWriter(fout)
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
    defer fout.Close()

    for _, row := range rows {
        writer.WriteString(row)
    }
    writer.Flush()
}
```

golang.fyi/ch10/bufwrite0.go

In general, the constructor functions in the `bufio` package create a buffered writer by wrapping an existing `io.Writer` as its underlying source. For instance, the previous code creates a buffered writer using the `bufio.NewWriter` function by wrapping the `io.File` variable, .

To influence the size of the internal buffer, use the constructor function `bufio.NewWriterSize(w io.Writer, n int)` to specify the internal buffer size. The `bufio.Writer` type also offers the methods `Write` and `WriteByte` for writing raw bytes and `WriteRune` for writing Unicode-encoded characters.

Reading buffered streams can be done simply by calling the constructor function `bufio.NewReader` to wrap an existing `io.Reader`. The following code snippet creates a `bufio.Reader` variable `reader` by wrapping the `file` variable as its underlying source:

```
func main() {
    file, err := os.Open("./bufread0.go")
    if err != nil {
        fmt.Println("Unable to open file:", err)
```

```
        return
    }
    defer file.Close()

    reader := bufio.NewReader(file)
    for {
        line, err := reader.ReadString('\n')
        if err != nil {
            if err == io.EOF {
                break
            } else {
                fmt.Println("Error reading:, err")
                return
            }
        }
        fmt.Print(line)
    }
}
```

golang.fyi/ch10/bufread0.go

The previous code uses the `reader.ReadString` method to read a text file using the '`\n`' character as the content delimiter. To influence the size of the internal buffer, use the constructor function `bufio.NewReaderSize(w io.Reader, n int)` to specify the internal buffer size. The `bufio.Reader` type also offers the `Read`, `ReadByte`, and `ReadBytes` methods for reading raw bytes from a stream and the `ReadRune` method for reading Unicode-encoded characters.

Scanning the buffer

The `bufio` package also makes available primitives that are used to scan and tokenize buffered input data from an `io.Reader` source. The `bufio.Scanner` type scans input data using the `Split` method to define tokenization strategies. The following code snippet shows a reimplementation of the planetary example (from earlier). This time, the code uses `bufio.Scanner` (instead of the `fmt.Fscan` function) to scan the content of the text file using the `bufio.ScanLines` function:

```
func main() {
    file, err := os.Open("./planets.txt")
    if err != nil {
        fmt.Println("Unable to open file:", err)
        return
    }
    defer file.Close()
```

```
    fmt.Printf(
        "%-10s %-10s %-6s %-6s\n",
        "Planet", "Diameter", "Moons", "Ring?",
    )
    scanner := bufio.NewScanner(file)
    scanner.Split(bufio.ScanLines)
    for scanner.Scan() {
        fields := strings.Split(scanner.Text(), " ")
        fmt.Printf(
            "%-10s %-10s %-6s %-6s\n",
            fields[0], fields[1], fields[2], fields[3],
        )
    }
}
```

golang.fyi/ch10/bufscan0.go

Using `bufio.Scanner` is done in four steps as shown in the previous example:

- First, use `bufio.NewScanner(io.Reader)` to create a scanner
- Call the `scanner.Split` method to configure how the content is tokenized
- Traverse the generated tokens with the `scanner.Scan` method
- Read the tokenized data with the `scanner.Text` method

The code uses the pre-defined function `bufio.ScanLines` to parse the buffered content using a line-delimiter. The `bufio` package comes with several pre-defined splitter functions including `ScanBytes` to scan each byte as a token, `ScanRunes` to scan UTF-8 encoded tokens, and `ScanWords` which scan each space-separated words as tokens.

In-memory IO

The `bytes` package offers common primitives to achieve streaming IO on blocks of bytes, stored in memory, represented by the `bytes.Buffer` type. Since the `bytes.Buffer` type implements both `io.Reader` and `io.Writer` interfaces it is a great option to stream data into or out of memory using streaming IO primitives.

The following snippet stores several string values in the `byte.Buffer` variable, `book`. Then the buffer is streamed to `os.Stdout`:

```
func main() {
    var books bytes.Buffer
    books.WriteString("The Great Gatsby")
    books.WriteString("1984")
```

```
    books.WriteString("A Tale of Two Cities")
    books.WriteString("Les Miserables")
    books.WriteString("The Call of the Wild")

    books.WriteTo(os.Stdout)
}
```

golang.fyi/ch10/bytesbuf0.go

The same example can easily be updated to stream the content to a regular file as shown in the following abbreviate code snippet:

```
func main() {
    var books bytes.Buffer
    books.WriteString("The Great Gatsby\n")
    books.WriteString("1984\n")
    books.WriteString("A Tale of Two Cities\n")
    books.WriteString("Les Miserables\n")
    books.WriteString("The Call of the Wild\n")

    file, err := os.Create("./books.txt")
    if err != nil {
        fmt.Println("Unable to create file:", err)
        return
    }
    defer file.Close()
    books.WriteTo(file)
}
```

golang.fyi/ch10/bytesbuf1.go

Encoding and decoding data

Another common aspect of IO in Go is the encoding of data, from one representation to another, as it is being streamed. The encoders and decoders of the standard library, found in the *encoding* package (<https://golang.org/pkg/encoding/>), use the `io.Reader` and `io.Writer` interfaces to leverage IO primitives as a way of streaming data during encoding and decoding.

Go supports several encoding formats for a variety of purposes including data conversion, data compaction, and data encryption. This chapter will focus on encoding and decoding data using the *Gob* and *JSON* format for data conversion. In Chapter 11, *Writing Networked Programs*, we will explore using encoders to convert data for client and server communication using **remote procedure calls (RPC)**.

Binary encoding with gob

The `gob` package (<https://golang.org/pkg/encoding/gob>) provides an encoding format that can be used to convert complex Go data types into binary. Gob is self-describing, meaning each encoded data item is accompanied by a type description. The encoding process involves streaming the gob-encoded data to an `io.Writer` so it can be written to a resource for future consumption.

The following snippet shows an example code that encodes variable `books`, a slice of the `Book` type with nested values, into the `gob` format. The encoder writes its generated binary data to an `os.Writer` instance, in this case the `file` variable of the `*os.File` type:

```
type Name struct {
    First, Last string
}

type Book struct {
    Title      string
    PageCount  int
    ISBN       string
    Authors    []Name
    Publisher  string
    PublishDate time.Time
}

func main() {
    books := []Book{
        Book{
            Title:      "Leaning Go",
            PageCount: 375,
            ISBN:       "9781784395438",
            Authors:    []Name{{"Vladimir", "Vivien"}},
            Publisher:  "Packt",
            PublishDate: time.Date(
                2016, time.July,
                0, 0, 0, 0, 0, time.UTC,
            ),
        },
        Book{
            Title:      "The Go Programming Language",
            PageCount: 380,
            ISBN:       "9780134190440",
            Authors:    []Name{
                {"Alan", "Donavan"},
                {"Brian", "Kernighan"},
            },
            Publisher:  "Addison-Wesley",
        },
    }
}
```

```
        PublishDate: time.Date(
            2015, time.October,
            26, 0, 0, 0, 0, time.UTC,
        ),
    },
    ...
}

// serialize data structure to file
file, err := os.Create("book.dat")
if err != nil {
    fmt.Println(err)
    return
}
enc := gob.NewEncoder(file)
if err := enc.Encode(books); err != nil {
    fmt.Println(err)
}
}
```

golang.fyi/ch10/gob0.go

Although the previous example is lengthy, it is mostly made of the definition of the nested data structure assigned to variable `books`. The last half-dozen or more lines are where the encoding takes place. The `gob` encoder is created with `enc := gob.NewEncoder(file)`. Encoding the data is done by simply calling `enc.Encode(books)` which streams the encoded data to the provide file.

The decoding process does the reverse by streaming the gob-encoded binary data using an `io.Reader` and automatically reconstructing it as a strongly-typed Go value. The following code snippet decodes the gob data that was encoded and stored in the `books.data` file in the previous example. The decoder reads the data from an `io.Reader`, in this instance the variable `file` of the `*os.File` type:

```
type Name struct {
    First, Last string
}

type Book struct {
    Title      string
    PageCount  int
    ISBN       string
    Authors    []Name
    Publisher  string
    PublishDate time.Time
}
```

```
func main() {
    file, err := os.Open("book.dat")
    if err != nil {
        fmt.Println(err)
        return
    }

    var books []Book
    dec := gob.NewDecoder(file)
    if err := dec.Decode(&books); err != nil {
        fmt.Println(err)
        return
    }
}
```

golang.fyi/ch10/gob1.go

Decoding a previously encoded gob data is done by creating a decoder using `dec := gob.NewDecoder(file)`. The next step is to declare the variable that will store the decoded data. In our example, the `books` variable, of the `[]Book` type, is declared as the destination of the decoded data. The actual decoding is done by invoking `dec.Decode(&books)`. Notice the `Decode()` method takes the address of its target variable as an argument. Once decoded, the `books` variable will contain the reconstituted data structure streamed from the file.



As of this writing, gob encoder and decoder APIs are only available in the Go programming language. This means that data encoded as gob can only be consumed by Go programs.

Encoding data as JSON

The encoding package also comes with a `json` encoder sub-package (<https://golang.org/pkg/encoding/json/>) to support JSON-formatted data. This greatly broadens the number of languages with which Go programs can exchange complex data structures. JSON encoding works similarly as the encoder and decoder from the `gob` package. The difference is that the generated data takes the form of a clear text JSON-encoded format instead of a binary. The following code updates the previous example to encode the data as JSON:

```
type Name struct {
    First, Last string
}

type Book struct {
```

```
Title      string
PageCount  int
ISBN       string
Authors    []Name
Publisher  string
PublishDate time.Time
}

func main() {
    books := []Book{
        Book{
            Title:      "Leaning Go",
            PageCount: 375,
            ISBN:       "9781784395438",
            Authors:    []Name{{"Vladimir", "Vivien"}},
            Publisher:  "Packt",
            PublishDate: time.Date(
                2016, time.July,
                0, 0, 0, 0, 0, time.UTC),
        },
        ...
    }

    file, err := os.Create("book.dat")
    if err != nil {
        fmt.Println(err)
        return
    }
    enc := json.NewEncoder(file)
    if err := enc.Encode(books); err != nil {
        fmt.Println(err)
    }
}
```

golang.fyi/ch10/json0.go

The code is exactly the same as before. It uses the same slice of nested structs assigned to the `books` variable. The only difference is the encoder is created with `enc := json.NewEncoder(file)` which creates a JSON encoder that will use the `file` variable as its `io.Writer` destination. When `enc.Encode(books)` is executed, the content of the variable `books` is serialized as JSON to the local file `books.dat`, shown in the following code (formatted for readability):

```
[  
  {  
    "Title": "Leaning Go",  
    "PageCount": 375,
```

```
        "ISBN": "9781784395438",
        "Authors": [{"First": "Vladimir", "Last": "Vivien"}],
        "Publisher": "Packt",
        "PublishDate": "2016-06-30T00:00:00Z"
    },
    {
        "Title": "The Go Programming Language",
        "PageCount": 380,
        "ISBN": "9780134190440",
        "Authors": [
            {"First": "Alan", "Last": "Donavan"},
            {"First": "Brian", "Last": "Kernighan"}
        ],
        "Publisher": "Addison-Wesley",
        "PublishDate": "2015-10-26T00:00:00Z"
    },
    ...
]
```

File books.dat (formatted)

The generated JSON-encoded content uses the name of the struct fields as the name for the JSON object keys by default. This behavior can be controlled using struct tags (see the section, *Controlling JSON mapping with struct tags*).

Consuming the JSON-encoded data in Go is done using a JSON decoder that streams its source from an `io.Reader`. The following snippet decodes the JSON-encoded data, generated in the previous example, stored in the file `book.dat`. Note that the data structure (not shown in the following code) is the same as before:

```
func main() {
    file, err := os.Open("book.dat")
    if err != nil {
        fmt.Println(err)
        return
    }

    var books []Book
    dec := json.NewDecoder(file)
    if err := dec.Decode(&books); err != nil {
        fmt.Println(err)
        return
    }
}
```

golang.fyi/ch10/json1.go

The data in the books.dat file is stored as an array of JSON objects. Therefore, the code must declare a variable capable of storing an indexed collection of nested struct values. In the previous example, the `books` variable, of the type `[]Book` is declared as the destination of the decoded data. The actual decoding is done by invoking `dec.Decode(&books)`. Notice the `Decode()` method takes the address of its target variable as an argument. Once decoded, the `books` variable will contain the reconstituted data structure streamed from the file.

Controlling JSON mapping with struct tags

By default, the name of a struct field is used as the key for the generated JSON object. This can be controlled using struct type tags to specify how JSON object key names are mapped during encoding and decoding of the data. For instance, the following code snippet declares struct fields with the `json:` tag prefix to specify how object keys are to be encoded and decoded:

```
type Book struct {
    Title      string      `json:"book_title"`
    PageCount  int         `json:"pages,string"`
    ISBN       string      `json:"-"`
    Authors    []Name      `json:"auths,omitempty"`
    Publisher  string      `json:",omitempty"`
    PublishDate time.Time `json:"pub_date"`
}
```

golang.fyi/ch10/json2.go

The tags and their meaning are summarized in the following table:

Tags	Description
<code>Title string `json:"book_title"</code>	Maps the <code>Title</code> struct field to the JSON object key, <code>"book_title"</code> .
<code>PageCount int `json:"pages,string"</code>	Maps the <code>PageCount</code> struct field to the JSON object key, <code>"pages"</code> , and outputs the value as a string instead of a number.
<code>ISBN string `json:"-"` </code>	The dash causes the <code>ISBN</code> field to be skipped during encoding and decoding.

Authors []Name `json:"auths,omitempty"`	Maps the Authors field to the JSON object key, "auths". The annotation, <code>omitempty</code> , causes the field to be omitted if its value is nil.
Publisher string `json:",omitempty"`	Maps the struct field name, Publisher, as the JSON object key name. The annotation, <code>omitempty</code> , causes the field to be omitted when empty.
PublishDate time.Time `json:"pub_date"`	Maps the field name, PublishDate, to the JSON object key, "pub_date".

When the previous struct is encoded, it produces the following JSON output in the books.dat file (formatted for readability):

```
...
{
    "book_title": "The Go Programming Language",
    "pages": "380",
    "auths": [
        {"First": "Alan", "Last": "Donavan"},
        {"First": "Brian", "Last": "Kernighan"}
    ],
    "Publisher": "Addison-Wesley",
    "pub_date": "2015-10-26T00:00:00Z"
}
...
```

Notice the JSON object keys are titled as specified in the struct tags. The object key "pages" (mapped to the struct field, `PageCount`) is encoded as a string. Finally, the struct field, `ISBN`, is omitted, as annotated in the struct tag.

Custom encoding and decoding

The JSON package uses two interfaces, `Marshaler` and `Unmarshaler`, to hook into encoding and decoding events respectively. When the encoder encounters a value whose type implements `json.Marshaler`, it delegates serialization of the value to the method `MarshalJSON` defined in the `Marshaller` interface. This is exemplified in the following abbreviated code snippet where the type `Name` is updated to implement `json.Marshaler` as shown:

```
type Name struct {
    First, Last string
}
```

```
func (n *Name) MarshalJSON() ([]byte, error) {
    return []byte(
        fmt.Sprintf("%s, %s", n.Last, n.First)
    ), nil
}

type Book struct {
    Title      string
    PageCount  int
    ISBN       string
    Authors    []Name
    Publisher  string
    PublishDate time.Time
}
func main() {
    books := []Book{
        Book{
            Title:      "Learning Go",
            PageCount: 375,
            ISBN:       "9781784395438",
            Authors:    []Name{{"Vladimir", "Vivien"}},
            Publisher:  "Packt",
            PublishDate: time.Date(
                2016, time.July,
                0, 0, 0, 0, 0, time.UTC),
        },
        ...
    }
    ...
    enc := json.NewEncoder(file)
    if err := enc.Encode(books); err != nil {
        fmt.Println(err)
    }
}
```

golang.fyi/ch10/json3.go

In the previous example, values of the `Name` type are serialized as a JSON string (instead of an object as earlier). The serialization is handled by the method `Name.MarshalJSON` which returns an array of bytes that contains the last and first name separated by a comma. The preceding code generates the following JSON output:

```
[  
  ...  
  {  
    "Title": "Learning Go",  
    "PageCount": 375,  
  }
```

```
        "ISBN": "9781784395438",
        "Authors": ["Vivien, Vladimir"],
        "Publisher": "Packt",
        "PublishDate": "2016-06-30T00:00:00Z"
    },
    ...
]
```

For the inverse, when a decoder encounters a piece of JSON text that maps to a type that implements `json.Unmarshaler`, it delegates the decoding to the type's `UnmarshalJSON` method. For instance, the following shows the abbreviated code snippet that implements `json.Unmarshaler` to handle the JSON output for the `Name` type:

```
type Name struct {
    First, Last string
}

func (n *Name) UnmarshalJSON(data []byte) error {
    var name string
    err := json.Unmarshal(data, &name)
    if err != nil {
        fmt.Println(err)
        return err
    }
    parts := strings.Split(name, ", ")
    n.Last, n.First = parts[0], parts[1]
    return nil
}
```

golang.fyi/ch10/json4.go

The `Name` type is an implementation of `json.Unmarshaler`. When the decoder encounters a JSON object with the key `"Authors"`, it uses the method `Name.Unmarshaler` to reconstitute the Go struct `Name` type from the JSON string.



The Go standard libraries offer additional encoders (not covered here) including `base32`, `base364`, `binary`, `csv`, `hex`, `xml`, `gzip`, and numerous encryption format encoders.

Summary

This chapter provides a high-level view of Go's data input and output idioms and the packages involved in implementing IO primitives. The chapter starts by covering the fundamentals of a stream-based IO in Go with the `io.Reader` and `io.Writer` interfaces. Readers are walked through the implementation strategies and examples for both an `io.Reader` and an `io.Writer`.

The chapter goes on to cover packages, types, and functions that support the streaming IO mechanism including working with files, formatted IO, buffered, and in-memory IO. The last portion of the chapter covers encoders and decoders that convert data as it is being streamed. In the next chapter, the IO theme is carried further when the discussion turns to creating programs that use IO to communicate via networking.

11

Writing Networked Services

One of the many reasons for Go's popularity, as a system language, is its inherent support for creating networked programs. The standard library exposes APIs ranging from low-level socket primitives to higher-level service abstractions such as HTTP and RPC. This chapter explores fundamental topics about creating connected applications including the following:

- The `net` package
- A TCP API server
- The `HTTP` package
- A JSON API server

The `net` package

The starting point for all networked programs in Go is the `net` package (<https://golang.org/pkg/net>). It provides a rich API to handle low-level networking primitives as well as application-level protocols such as HTTP. Each logical component of a network is represented by a Go type including hardware interfaces, networks, packets, addresses, protocols, and connections. Furthermore, each type exposes a multitude of methods giving Go one of the most complete standard libraries for network programming supporting both IPv4 and IPv6.

Whether creating a client or a server program, Go programmers will need, at a minimum, the network primitives covered in the following sections. These primitives are offered as functions and types to facilitate clients connecting to remote services and servers to handle incoming requests.

Addressing

One of the basic primitives, when doing network programming, is the *address*. The types and functions of the `net` package use a string literal to represent an address such as `"127.0.0.1"`. The address can also include a service port separated by a colon such as `"74.125.21.113:80"`. Functions and methods in the `net` package also support string literal representation for IPv6 addresses such as `"::1"` or `"[2607:f8b0:4002:c06::65]:80"` for an address with a service port of 80.

The `net.Conn` Type

The `net.Conn` interface represents a generic connection established between two nodes on the network. It implements `io.Reader` and `io.Writer` interfaces which allow connected nodes to exchange data using streaming IO primitives. The `net` package offers network protocol-specific implementations of the `net.Conn` interface such as `IPConn`, `UDPConn`, and `TCPConn`. Each implementation exposes additional methods specific to its respective network and protocol. However, as we will see in this chapter, the default method set defined in `net.Conn` is adequate for most uses.

Dialing a connection

Client programs use the `net.Dial` function, which has the following signature, to connect to a host service over the network:

```
func Dial(network, address string) (Conn, error)
```

The function takes two parameters where the first parameter, `network`, specifies the network protocol for the connection which can be:

- `tcp`, `tcp4`, `tcp6`: `tcp` defaults to `tcp4`
- `udp`, `udp4`, `udp6`: `udp` defaults to `udp4`
- `ip`, `ip4`, `ip6`: `ip` defaults to `ip4`
- `unix`, `unixgram`, `unixpacket`: for Unix domain sockets

The latter parameter of the `net.Dial` function specifies a string value for the host address to which to connect. The address can be provided as IPv4 or IPv6 addresses as discussed earlier. The `net.Dial` function returns an implementation of the `net.Conn` interface that matches the specified network parameter.

For instance, the following code snippet dials a "tcp" network at the host address, `www.gutenberg.org:80`, which returns a TCP connection of the `*net.TCPConn` type. The abbreviated code uses the TCP connection to issue an "HTTP GET" request to retrieve the full text of the literary classic Beowulf from the Project Gutenberg's website (`http://gutenberg.org/`). The raw and unparsed HTTP response is subsequently written to a local file, `beowulf.txt`:

```
func main() {
    host, port := "www.gutenberg.org", "80"
    addr := net.JoinHostPort(host, port)
    httpRequest := "GET /cache/epub/16328/pg16328.txt HTTP/1.1\n" +
        "Host: " + host + "\n\n"

    conn, err := net.Dial("tcp", addr)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer conn.Close()

    if _, err = conn.Write([]byte(httpRequest)); err != nil {
        fmt.Println(err)
        return
    }

    file, err := os.Create("beowulf.txt")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer file.Close()

    io.Copy(file, conn)
    fmt.Println("Text copied to file", file.Name())
}
```

`golang.fyi/ch11/dial0.go`

Because the `net.Conn` type implements the `io.Reader` and `io.Writer`, it can be used to both send data and receive data using streaming IO semantics. In the preceding example, `conn.Write([]byte(httpRequest))` sends the HTTP request to the server. The response returned by the host is copied from the `conn` variable to the `file` variable using `io.Copy(file, conn)`.



Note that the previous is an illustration that shows how to connect to an HTTP server using raw TCP. The Go standard library provides a separate package designed specifically for HTTP programming which abstracts away the low-level protocol details (covered later in the chapter).

The `net` package also makes available network specific dialing functions such as `DialUDP`, `DiapTCP`, or `DialIP`, each returning its respective connection implementation. In most cases, the `net.Dial` function and the `net.Conn` interface provide adequate capabilities to connect and manage connections to a remote host.

Listening for incoming connections

When creating a service program, one of the first steps is to announce the port which the service will use to listen for incoming requests from the network. This is done by invoking the `net.Listen` function which has the following signature:

```
func Listen(network, laddr string) (net.Listener, error)
```

It takes two parameters where the first parameter specifies a protocol with valid values of `"tcp"`, `"tcp4"`, `"tcp6"`, `"unix"`, or `"unixpacket"`.

The second parameter is the local host address for the service. The local address can be specified without an IP address such as ":4040". Omitting the IP address of the host means that the service is bound to all network card interfaces installed on the host. As an alternative, the service can be bound to a specific network hardware interface on the host by specifying its IP address on the network, that is, `"10.20.130.240:4040"`.

A successful call to the `net.Listen` function returns a value of the `net.Listener` type (or a non-nil error if it fails). The `net.Listener` interface exposes methods used to manage the life cycle of incoming client connections. Depending on the value of the `network` parameter (`"tcp"`, `"tcp4"`, `"tcp6"`, and so on.), `net.Listen` will return either a `net.TCPListener` or `net.UnixListener`, both of which are concrete implementations of the `net.Listener` interface.

Accepting client connections

The `net.Listener` interface uses the `Accept` method to block indefinitely until a new connection arrives from a client. The following abbreviated code snippet shows a simple server that returns the string "Nice to meet you!" to each client connection and then disconnects immediately:

```
func main() {
    listener, err := net.Listen("tcp", ":4040")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer listener.Close()

    for {
        conn, err := listener.Accept()
        if err != nil {
            fmt.Println(err)
            return
        }
        conn.Write([]byte("Nice to meet you!"))
        conn.Close()
    }
}
```

golang.fyi/ch11/listen0.go

In the code, the `listener.Accept` method returns a value of the `net.Conn` type to handle data exchange between the server and the client (or it returns a non-nil `error` if it fails). The `conn.Write([]byte("Nice to meet you!"))` method call is used to write the response to the client. When the server program is running, it can be tested using a *telnet* client as shown in the following output:

```
$> go run listen0.go &
[1] 83884

$> telnet 127.0.0.1 4040
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^].
Nice to meet you! Connection closed by foreign host.
```

To ensure that the server program continues to run and handle subsequent client connections, the call to the `Accept` method is wrapped within an infinite for-loop. As soon as a connection is closed, the loop restarts the cycle to wait for the next client connection. Also notice that it is a good practice to close the listener when the server process is shutting down with a call to `Listener.Close()`.



The observant reader may notice that this simple server will not scale as it cannot handle more than one client request at once. In the next section, we will see the techniques for creating a scalable server.

A TCP API server

At this point, the chapter has covered the minimum networking components necessary to create client and service programs. The remainder of the chapter will discuss different versions of a server that implement a *monetary currency information* service. The service returns ISO 4217 monetary currency information with each request. The intent is to show the implications of creating networked services, along with their clients, using different application-level protocols.

Earlier we introduced a very simple server to demonstrate the necessary steps required to set up a networked service. This section dives deeper into network programming by creating a TCP server that scales to handle many concurrent connections. The server code presented in this section has the following design goals:

- Use raw TCP to communicate between client and server
- Develop a simple text-based protocol, over TCP, for communication
- Clients can query the server for global currency information with text commands
- Use a goroutine per connection to handle connection concurrency
- Maintain connection until the client disconnects

The following lists an abbreviated version of the server code. The program uses the `curr` package (found at <https://github.com/vladimirvivien/learning-go/ch11/curr0>), not discussed here, to load monetary currency data from a local CSV file into slice currencies.

Upon successful connection to a client, the server parses the incoming client commands specified with a simple text protocol with the format *GET <currency-filter-value>* where *<currency-filter-value>* specifies a string value used to search for currency information:

```
import (
    "net"
    ...
    curr "https://github.com/vladimirvivien/learning-go/ch11/curr0"
)

var currencies = curr.Load("./data.csv")

func main() {
    ln, _ := net.Listen("tcp", ":4040")
    defer ln.Close()
    // connection loop
    for {
        conn, err := ln.Accept()
        if err != nil {
            fmt.Println(err)
            conn.Close()
            continue
        }
        go handleConnection(conn)
    }
}

// handle client connection
func handleConnection(conn net.Conn) {
    defer conn.Close()

    // loop to stay connected with client
    for {
        cmdLine := make([]byte, (1024 * 4))
        n, err := conn.Read(cmdLine)
        if n == 0 || err != nil {
            return
        }
        cmd, param := parseCommand(string(cmdLine[0:n]))
        if cmd == "" {
            continue
        }

        // execute command
        switch strings.ToUpper(cmd) {
        case "GET":
            result := curr.Find(currencies, param)
            // stream result to client
        }
    }
}
```

```
        for _, cur := range result {
            _, err := fmt.Fprintf(
                conn,
                "%s %s %s %s\n",
                cur.Name, cur.Code,
                cur.Number, cur.Country,
            )
            if err != nil {
                return
            }
            // reset deadline while writing,
            // closes conn if client is gone
            conn.SetWriteDeadline(
                time.Now().Add(time.Second * 5))
        }
        // reset read deadline for next read
        conn.SetReadDeadline(
            time.Now().Add(time.Second * 300))

    default:
        conn.Write([]byte("Invalid command\n"))
    }
}

func parseCommand(cmdLine string) (cmd, param string) {
    parts := strings.Split(cmdLine, " ")
    if len(parts) != 2 {
        return "", ""
    }
    cmd = strings.TrimSpace(parts[0])
    param = strings.TrimSpace(parts[1])
    return
}
```

golang.fyi/ch11/tcpserv0.go

Unlike the simple server introduced in the last section, this server is able to service multiple client connections at the same time. Upon accepting a new connection, with `ln.Accept()`, it delegates the handling of new client connections to a goroutine with `go handleConnection(conn)`. The connection loop then continues immediately and waits for the next client connection.

The `handleConnection` function manages the server communication with the connected client. It first reads and parses a slice of bytes, from the client, into a command string using `cmd, param := parseCommand(string(cmdLine[0:n]))`. Next, the code tests the command with a `switch` statement. If the `cmd` is equal to "GET", the code searches slice `currencies` for values that matches `param` with a call to `curr.Find(currencies, param)`. Finally, it streams the search result to the client's connection using `fmt.Fprintf(conn, "%s %s %s %s\n", cur.Name, cur.Code, cur.Number, cur.Country)`.

The simple text protocol supported by the server does not include any sort of session control or control messages. Therefore, the code uses the `conn.SetWriteDeadline` method to ensure the connection to the client does not linger unnecessarily for long periods of time. The method is called during the loop that streams out a response to the client. It is set for a deadline of 5 seconds to ensure the client is always ready to receive the next chunk of bytes within that time, otherwise it times the connection out.

Connecting to the TCP server with telnet

Because the currency server presented earlier uses a simple text-based protocol, it can be tested using a telnet client, assuming the server code has been compiled and running (and listening on port 4040). The following shows the output of a telnet session querying the server for currency information:

```
$> telnet localhost 4040
Trying ::1...
Connected to localhost.
Escape character is '^]'.
GET Gourde
Gourde HTG 332 HAITI
GET USD
US Dollar USD 840 AMERICAN SAMOA
US Dollar USD 840 BONAIRE, SINT EUSTATIUS AND SABA
US Dollar USD 840 GUAM
US Dollar USD 840 HAITI
US Dollar USD 840 MARSHALL ISLANDS (THE)
US Dollar USD 840 UNITED STATES OF AMERICA (THE)
...
```

```
get india
Indian Rupee INR 356 BHUTAN
US Dollar USD 840 BRITISH INDIAN OCEAN TERRITORY (THE)
Indian Rupee INR 356 INDIA
```

As you can see, you can query the server by using the `get` command followed by a filter parameter as explained earlier. The telnet client sends the raw text to the server which parses it and sends back raw text as the response. You can open multiple telnet sessions against the server and all request are served concurrently in their respective goroutine.

Connecting to the TCP server with Go

A simple TCP client can also be written in Go to connect to the TCP server. The client captures the command from the console's standard input and sends it to the server as is shown in the following code snippet:

```
var host, port = "127.0.0.1", "4040"
var addr = net.JoinHostPort(host, port)
const prompt = "curr"
const buffLen = 1024

func main() {
    conn, err := net.Dial("tcp", addr)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer conn.Close()
    var cmd, param string
    // repl - interactive shell for client
    for {
        fmt.Print(prompt, "> ")
        _, err = fmt.Scanf("%s %s", &cmd, &param)
        if err != nil {
            fmt.Println("Usage: GET <search string or *>")
            continue
        }
        // send command line
        cmdLine := fmt.Sprintf("%s %s", cmd, param)
        if n, err := conn.Write([]byte(cmdLine));
        n == 0 || err != nil {
            fmt.Println(err)
            return
        }

        // stream and display response
```

```
        conn.SetReadDeadline(
            time.Now().Add(time.Second * 5))
    for {
        buff := make([]byte, buffLen)
        n, err := conn.Read(buff)
        if err != nil { break }
        fmt.Println(string(buff[0:n]))
        conn.SetReadDeadline(
            time.Now().Add(time.Millisecond * 700))
    }
}
```

golang.fyi/ch11/tcpclient0.go

The source code for the Go client follows the same pattern as we have seen in the earlier client example. The first portion of the code dials out to the server using `net.Dial()`. Once a connection is obtained, the code sets up an event loop to capture text commands from the standard input, parses it, and sends it as a request to the server.

There is a nested loop that is set up to handle incoming responses from the server (see code comment). It continuously streams incoming bytes into variables `buff` with `conn.Read(buff)`. This continues until the `Read` method encounters an error. The following lists the sample output produced by the client when it is executed:

```
$> Connected to Global Currency Service
curr> get pound
Egyptian Pound EGP 818 EGYPT
Gibraltar Pound GIP 292 GIBRALTAR
Sudanese Pound SDG 938 SUDAN (THE)
...
Syrian Pound SYP 760 SYRIAN ARAB REPUBLIC
Pound Sterling GBP 826 UNITED KINGDOM OF GREAT BRITAIN (THE)
curr>
```

An even better way of streaming the incoming bytes from the server is to use buffered IO as done in the following snippet of code. In the updated code, the `conbuf` variable, of the `bufio.Buffer` type, is used to read and split incoming streams from the server using the `conbuf.ReadString` method:

```
conbuf := bufio.NewReaderSize(conn, 1024)
for {
    str, err := conbuf.ReadString('\n')
    if err != nil {
        break
    }
}
```

```
        fmt.Println(str)
        conn.SetReadDeadline(
            time.Now().Add(time.Millisecond * 700))
    }

golang.fyi/ch11/tcpclient1.go
```

As you can see, writing networked services directly on top of raw TCP has some costs. While raw TCP gives the programmer complete control of the application-level protocol, it also requires the programmer to carefully handle all data processing which can be error-prone. Unless it is absolutely necessary to implement your own custom protocol, a better approach is to leverage an existing and proven protocols to implement your server programs. The remainder of this chapter continues to explore this topic using services that are based on HTTP as an application-level protocol.

The HTTP package

Due to its importance and ubiquity, HTTP is one of a handful of protocols directly implemented in Go. The `net/http` package (<https://golang.org/pkg/net/http/>) provides code to implement both HTTP clients and HTTP servers. This section explores the fundamentals of creating HTTP clients and servers using the `net/http` package. Later, we will return our attention back to building versions of our currency service using HTTP.

The `http.Client` type

The `http.Client` struct represents an HTTP client and is used to create HTTP requests and retrieve responses from a server. The following illustrates how to retrieve the text content of Beowulf from Project Gutenberg's website located at <http://gutenberg.org/cache/epub/16328/pg16328.txt>, using the `client` variable of the `http.Client` type and prints its content to a standard output:

```
func main() {
    client := http.Client{}
    resp, err := client.Get(
        "http://gutenberg.org/cache/epub/16328/pg16328.txt")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer resp.Body.Close()
    io.Copy(os.Stdout, resp.Body)
```

```
}
```

```
golang.fyi/ch11/httpclient1.go
```

The previous example uses the `client.Get` method to retrieve content from the remote server using the HTTP protocol method `GET` internally. The `GET` method is part of several convenience methods offered, by the `Client` type, to interact with HTTP servers as summarized in the following table. Notice that all of these methods return a value of the `*http.Response` type (discussed later) to handle responses returned by the HTTP server.

Method	Description
<code>Client.Get</code>	As discussed earlier, <code>Get</code> is a convenience method that issues an HTTP <code>GET</code> method to retrieve the resource specified by the <code>url</code> parameter from the server: <code>Get(url string,) (resp *http.Response, err error)</code>
<code>Client.Post</code>	The <code>Post</code> method is a convenience method that issues an HTTP <code>POST</code> method to send the content specified by the <code>body</code> parameter to the server specified by the <code>url</code> parameter: <code>Post(url string, bodyType string, body io.Reader,) (resp *http.Response, err error)</code>
<code>Client.PostForm</code>	The <code>PostForm</code> method is a convenience method that uses the HTTP <code>POST</code> method to send form data, specified as mapped key/value pairs, to the server: <code>PostForm(url string, data url.Values,) (resp *http.Response, err error)</code>
<code>Client.Head</code>	The <code>Head</code> method is a convenience method that issues an HTTP method, <code>HEAD</code> , to the remote server specified by the <code>url</code> parameter: <code>Head(url string,) (resp *http.Response, err error)</code>
<code>Client.Do</code>	This method generalizes the request and response interaction with a remote HTTP server. It is wrapped internally by the methods listed in this table. Section <i>Handling client requests and responses</i> discusses how to use this method to talk to the server.

It should be noted that the HTTP package uses an internal `http.Client` variable designed to mirror the preceding methods as package functions for further convenience. They include `http.Get`, `http.Post`, `http.PostForm`, and `http.Head`. The following snippet shows the previous example using `http.Get` instead of the method from the `http.Client`:

```
func main() {
    resp, err := http.Get(
        "http://gutenberg.org/cache/epub/16328/pg16328.txt")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer resp.Body.Close()
    io.Copy(os.Stdout, resp.Body)
}
```

golang.fyi/ch11/httpclient1a.go

Configuring the client

Besides the methods to communicate with the remote server, the `http.Client` type exposes additional attributes that can be used to modify and control the behavior of the client. For instance, the following source snippet sets the timeout to handle a client request to complete within 21 seconds using the `Timeout` attribute of the `Client` type:

```
func main() {
    client := &http.Client{
        Timeout: 21 * time.Second
    }
    resp, err := client.Get(
        "http://tools.ietf.org/rfc/rfc7540.txt")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer resp.Body.Close()
    io.Copy(os.Stdout, resp.Body)
}
```

golang.fyi/ch11/httpclient2.go

The `Transport` field of the `Client` type provides further means of controlling the settings of a client. For instance, the following snippet creates a client that disables the connection reuse between successive HTTP requests with the `DisableKeepAlive` field. The code also uses the `Dial` function to specify further granular control over the HTTP connection used by the underlying client, setting its timeout value to 30 seconds:

```
func main() {
    client := &http.Client{
        Transport: &http.Transport{
            DisableKeepAlives: true,
            Dial: (&net.Dialer{
                Timeout: 30 * time.Second,
            }).Dial,
        },
    }
    ...
}
```

Handling client requests and responses

An `http.Request` value can be explicitly created using the `http.NewRequest` function. A request value can be used to configure HTTP settings, add headers, and specify the content body of the request. The following source snippet uses the `http.Request` type to create a new request which is used to specify the headers sent to the server:

```
func main() {
    client := &http.Client{}
    req, err := http.NewRequest(
        "GET", "http://tools.ietf.org/rfc/rfc7540.txt", nil,
    )
    req.Header.Add("Accept", "text/plain")
    req.Header.Add("User-Agent", "SampleClient/1.0")

    resp, err := client.Do(req)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer resp.Body.Close()
    io.Copy(os.Stdout, resp.Body)
}
```

golang.fyi/ch11/httpclient3.go

The `http.NewRequest` function has the following signature:

```
func NewRequest(method, uStr string, body io.Reader) (*http.Request, error)
```

It takes a string that specifies the HTTP method as its first argument. The next argument specifies the destination URL. The last argument is an `io.Reader` that can be used to specify the content of the request (or set to `nil` if the request has no content). The function returns a pointer to a `http.Request` struct value (or a non-`nil` `error` if one occurs). Once the request value is created, the code uses the `Header` field to add HTTP headers to the request to be sent to the server.

Once a request is prepared (as shown in the previous source snippet), it is sent to the server using the `Do` method of the `http.Client` type and has the following signature:

```
Do(req *http.Request) (*http.Response, error)
```

The method accepts a pointer to an `http.Request` value, as discussed in the previous section. It then returns a pointer to an `http.Response` value or an `error` if the request fails. In the previous source code, `resp, err := client.Do(req)` is used to send the request to the server and assigns the response to the `resp` variable.

The response from the server is encapsulated in struct `http.Response` which contains several fields to describe the response including the HTTP response status, content length, headers, and the response body. The response body, exposed as the `http.Response.Body` field, implements the `io.Reader` which affords the use streaming IO primitives to consume the response content.

The `Body` field also implements `io.Closer` which allows the closing of IO resources. The previous source uses `defer resp.Body.Close()` to close the IO resource associated with the response body. This is a recommended idiom when the server is expected to return a non-`nil` body.

A simple HTTP server

The HTTP package provides two main components to accept HTTP requests and serve responses:

- The `http.Handler` interface
- The `http.Server` type

The `http.Server` type uses the `http.Handler` interface type, defined in the following listing, to receive requests and server responses:

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

Any type that implements `http.Handler` can be registered (explained next) as a valid handler. The Go `http.Server` type is used to create a new server. It is a struct whose values can be configured, at a minimum, with the TCP address of the service and a handler that will respond to incoming requests. The following code snippet shows a simple HTTP server that defines the `msg` type as handler registered to handle incoming client requests:

```
type msg string

func (m msg) ServeHTTP(
    resp http.ResponseWriter, req *http.Request) {
    resp.Header().Add("Content-Type", "text/html")
    resp.WriteHeader(http.StatusOK)
    fmt.Fprint(resp, m)
}

func main() {
    msgHandler := msg("Hello from high above!")
    server := http.Server{Addr: ":4040", Handler: msgHandler}
    server.ListenAndServe()
}
```

golang.fyi/ch11/httpserv0.go

In the previous code, the `msg` type, which uses a string as its underlying type, implements the `ServeHTTP()` method making it a valid HTTP handler. Its `ServeHTTP` method uses the `response` parameter, `resp`, to print response headers "200 OK" and "Content-Type: text/html". The method also writes the string value `m` to the response variable using `fmt.Fprint(resp, m)` which is sent back to the client.

In the code, the variable `server` is initialized as `http.Server{Addr: ":4040", Handler: msgHandler}`. This means the server will listen on all network interfaces at port 4040 and will use variable `msgHandler` as its `http.Handler` implementation. Once initialized, the server is started with the `server.ListenAndServe()` method call that is used to block and listen for incoming requests.

Besides the `Addr` and `Handler`, the `http.Server` struct exposes several additional fields that can be used to control different aspects of the HTTP service such as connection, timeout values, header sizes, and TLS configuration. For instance, the following snippet shows an updated example which specifies the server's read and write timeouts:

```
type msg string
func (m msg) ServeHTTP(
    resp http.ResponseWriter, req *http.Request) {
    resp.Header().Add("Content-Type", "text/html")
    resp.WriteHeader(http.StatusOK)
    fmt.Fprint(resp, m)
}
func main() {
    msgHandler := msg("Hello from high above!")
    server := http.Server{
        Addr:          ":4040",
        Handler:       msgHandler,
        ReadTimeout:   time.Second * 5,
        WriteTimeout:  time.Second * 3,
    }
    server.ListenAndServe()
}
```

golang.fyi/ch11/httpserv1.go

The default server

It should be noted that the `HTTP` package includes a default server that can be used in simpler cases when there is no need for configuration of the server. The following abbreviated code snippet starts a simple server without explicitly creating a `server` variable:

```
type msg string

func (m msg) ServeHTTP(
    resp http.ResponseWriter, req *http.Request) {
    resp.Header().Add("Content-Type", "text/html")
    resp.WriteHeader(http.StatusOK)
    fmt.Fprint(resp, m)
}
func main() {
    msgHandler := msg("Hello from high above!")
    http.ListenAndServe(":4040", msgHandler)
}
```

golang.fyi/ch11/httpserv2.go

In the code, the `http.ListenAndServe(":4040", msgHandler)` function is used to start a server which is declared as a variable in the HTTP package. The server is configured with the local address "`:4040`" and the handler `msgHandler` (as was done earlier) to handle all incoming requests.

Routing requests with `http.ServeMux`

The `http.Handler` implementation introduced in the previous section is not sophisticated. No matter what URL path is sent with the request, it sends the same response back to the client. That is not very useful. In most cases, you want to map each path of a request URL to a different response.

Fortunately, the HTTP package comes with the `http.ServeMux` type which can multiplex incoming requests based on URL patterns. When an `http.ServeMux` handler receives a request, associated with a URL path, it dispatches a function that is mapped to that URL. The following abbreviated code snippet shows `http.ServeMux` variable `mux` configured to handle two URL paths `"/hello"` and `"/goodbye"`:

```
func main() {
    mux := http.NewServeMux()
    hello := func(resp http.ResponseWriter, req *http.Request) {
        resp.Header().Add("Content-Type", "text/html")
        resp.WriteHeader(http.StatusOK)
        fmt.Fprint(resp, "Hello from Above!")
    }

    goodbye := func(resp http.ResponseWriter, req *http.Request) {
        resp.Header().Add("Content-Type", "text/html")
        resp.WriteHeader(http.StatusOK)
        fmt.Fprint(resp, "Goodbye, it's been real!")
    }

    mux.HandleFunc("/hello", hello)
    mux.HandleFunc("/goodbye", goodbye)

    http.ListenAndServe(":4040", mux)
}
```

golang.fyi/ch11/httpserv3.go

The code declares two functions assigned to variables `hello` and `goodbye`. Each function is mapped to a path `"/hello"` and `"/goodbye"` respectively using the `mux.HandleFunc("/hello", hello)` and `mux.HandleFunc("/goodbye", goodbye)` method calls. When the server is launched, with `http.ListenAndServe(":4040", mux)`, its handler will route the request `"http://localhost:4040/hello"` to the `hello` function and requests with the path `"http://localhost:4040/goodbye"` to the `goodbye` function.

The default ServeMux

It is worth pointing out that the HTTP package makes available a default `ServeMux` internally. When used, it is not necessary to explicitly declare a `ServeMux` variable. Instead the code uses the package function, `http.HandleFunc`, to map a path to a handler function as illustrated in the following code snippet:

```
func main() {
    hello := func(resp http.ResponseWriter, req *http.Request) {
        ...
    }

    goodbye := func(resp http.ResponseWriter, req *http.Request) {
        ...
    }

    http.HandleFunc("/hello", hello)
    http.HandleFunc("/goodbye", goodbye)

    http.ListenAndServe(":4040", nil)
}
```

golang.fyi/ch11/httpserv4.go

To launch the server, the code calls `http.ListenAndServe(":4040", nil)` where the `ServerMux` parameter is set to `nil`. This implies that the server will default to the pre-declared package instance of `http.ServeMux` to handle incoming requests.

A JSON API server

Armed with the information from the last section, it is possible to use the HTTP package to create services over HTTP. Earlier we discussed the perils of creating services using raw TCP directly when we created a server for our global monetary currency service. In this section, we explore how to create an API server for the same service using HTTP as the underlying protocol. The new HTTP-based service has the following design goals:

- Use HTTP as the transport protocol
- Use JSON for structured communication between client and server
- Clients query the server for currency information using JSON-formatted requests
- The server respond using JSON-formatted responses

The following shows the code involved in the implementation of the new service. This time, the server will use the `curr1` package (see github.com/vladimirvivien/learning-go/ch11/curr1) to load and query ISO 4217 currency data from a local CSV file.

The code in the `curr1` package defines two types, `CurrencyRequest` and `Currency`, intended to represent the client request and currency data returned by the server, respectively as listed here:

```
type Currency struct {
    Code    string `json:"currency_code"`
    Name    string `json:"currency_name"`
    Number  string `json:"currency_number"`
    Country string `json:"currency_country"`
}

type CurrencyRequest struct {
    Get    string `json:"get"`
    Limit int    `json:limit`
}
```

golang.fyi/ch11/curr1/currency.go

Note that the preceding struct types shown are annotated with tags that describe the JSON properties for each field. This information is used by the JSON encoder to encode the key name of JSON objects (see Chapter 10, *Data IO in Go*, for detail on encoding). The remainder of the code, listed in the following snippet, defines the functions that set up the server and the handler function for incoming requests:

```
import (
    "encoding/json"
    "fmt"
```

```
"net/http"
" github.com/vladimirvivien/learning-go/ch11/curr1"
)
var currencies = curr1.Load("./data.csv")

func currs(resp http.ResponseWriter, req *http.Request) {
    var currRequest curr1.CurrencyRequest
    dec := json.NewDecoder(req.Body)
    if err := dec.Decode(&currRequest); err != nil {
        resp.WriteHeader(http.StatusBadRequest)
        fmt.Println(err)
        return
    }

    result := curr1.Find(currencies, currRequest.Get)
    enc := json.NewEncoder(resp)
    if err := enc.Encode(&result); err != nil {
        fmt.Println(err)
        resp.WriteHeader(http.StatusInternalServerError)
        return
    }
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/currency", get)

    if err := http.ListenAndServe(":4040", mux); err != nil {
        fmt.Println(err)
    }
}
```

golang.fyi/ch11/jsonserv0.go

Since we are leveraging HTTP as the transport protocol for the service, you can see the code is now much smaller than the prior implementation which used pure TCP. The `currs` function implements the handler responsible for incoming requests. It sets up a decoder to decode the incoming JSON-encoded request to a value of the `curr1.CurrencyRequest` type as highlighted in the following snippet:

```
var currRequest curr1.CurrencyRequest
dec := json.NewDecoder(req.Body)
if err := dec.Decode(&currRequest); err != nil { ... }
```

Next, the function executes the currency search by calling `curr1.Find(currencies, currRequest.Get)` which returns the slice `[]Currency` assigned to the `result` variable. The code then creates an encoder to encode the `result` as a JSON payload, highlighted in the following snippet:

```
result := curr1.Find(currencies, currRequest.Get)
enc := json.NewEncoder(resp)
if err := enc.Encode(&result); err != nil { ... }
```

Lastly, the handler function is mapped to the `"/currency"` path in the `main` function with the call to `mux.HandleFunc("/currency", curr)`. When the server receives a request for that path, it automatically executes the `curr` function.

Testing the API server with cURL

Because the server is implemented over HTTP, it can easily be tested with any client-side tools that support HTTP. For instance, the following shows how to use the `cURL` command line tool (`http://curl.haxx.se/`) to connect to the API end-point and retrieve currency information about the Euro:

```
$> curl -X POST -d '{"get":"Euro"}' http://localhost:4040/currency
[
  ...
  {
    "currency_code": "EUR",
    "currency_name": "Euro",
    "currency_number": "978",
    "currency_country": "BELGIUM"
  },
  {
    "currency_code": "EUR",
    "currency_name": "Euro",
    "currency_number": "978",
    "currency_country": "FINLAND"
  },
  {
    "currency_code": "EUR",
    "currency_name": "Euro",
    "currency_number": "978",
    "currency_country": "FRANCE"
  },
  ...
]
```

The `CURL` command posts a JSON-formatted request object to the server using the `-X POST -d '{"get": "Euro"}'` parameters. The output (formatted for readability) from the server is comprised of a JSON array of the preceding currency items.

An API server client in Go

An HTTP client can also be built in Go to consume the service with minimal efforts. As is shown in the following code snippet, the client code uses the `http.Client` type to communicate with the server. It also uses the `encoding/json` sub-package to decode incoming data (note that the client also makes use of the `curr1` package, shown earlier, which contains the types needed to communicate with the server):

```
import (
    "bytes"
    "encoding/json"
    "fmt"
    "net/http"

    "github.com/vladimirvivien/learning-go/ch11/curr1"
)

func main() {
    var param string
    fmt.Print("Currency> ")
    _, err := fmt.Scanf("%s", &param)

    buf := new(bytes.Buffer)
    currRequest := &curr1.CurrencyRequest{Get: param}
    err = json.NewEncoder(buf).Encode(currRequest)
    if err != nil {
        fmt.Println(err)
        return
    }

    // send request
    client := &http.Client{}
    req, err := http.NewRequest(
        "POST", "http://127.0.0.1:4040/currency", buf)
    if err != nil {
        fmt.Println(err)
        return
    }

    resp, err := client.Do(req)
    if err != nil {
```

```
        fmt.Println(err)
        return
    }
    defer resp.Body.Close()

    // decode response
    var currencies []curr1.Currency
    err = json.NewDecoder(resp.Body).Decode(&currencies)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(currencies)
}
```

golang.fyi/ch11/jsonclient0.go

In the previous code, an HTTP client is created to send JSON-encoded request values as `currRequest := &curr1.CurrencyRequest{Get: param}` where `param` is the currency string to retrieve. The response from the server is a payload that represents an array of JSON-encoded objects (see the JSON array in the section, *Testing the API Server with cURL*). The code then uses a JSON decoder, `json.NewDecoder(resp.Body).Decode(¤cies)`, to decode the payload from the response body into the slice, `[]curr1.Currency`.

A JavaScript API server client

So far, we have seen how to use the API service using the `cURL` command-line tool and a native Go client. This section shows the versatility of using HTTP to implement networked services by showcasing a web-based JavaScript client. In this approach, the client is a web-based GUI that uses modern HTML, CSS, and JavaScript to create an interface that interacts with the API server.

First, the server code is updated with an additional handler to serve the static HTML file that renders the GUI on the browser. This is illustrated in the following code:

```
// serves HTML gui
func gui(resp http.ResponseWriter, req *http.Request) {
    file, err := os.Open("./currency.html")
    if err != nil {
        resp.WriteHeader(http.StatusInternalServerError)
        fmt.Println(err)
        return
    }
```

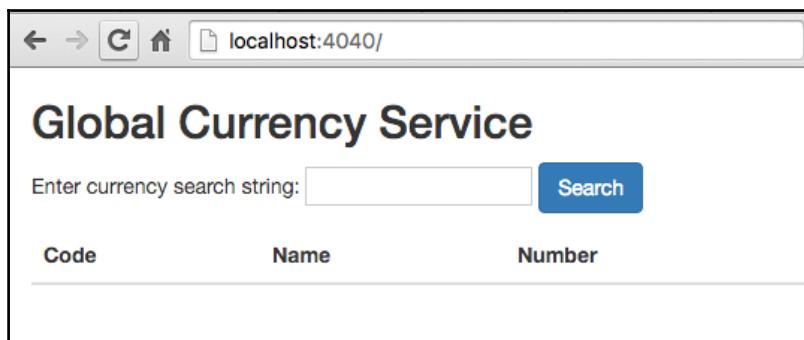
```
    io.Copy(resp, file)
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", gui)
    mux.HandleFunc("/currency", currs)

    if err := http.ListenAndServe(":4040", mux); err != nil {
        fmt.Println(err)
    }
}
```

golang.fyi/ch11/jsonserv1.go

The preceding code snippet shows the declaration of the `gui` handler function responsible for serving a static HTML file that renders the GUI for the client. The root URL path is then mapped to the function with `mux.HandleFunc("/", gui)`. So, in addition to the `"/currency"` path, which hosts the API end-point the `"/"` path will return the web page shown in the following screenshot:



The next HTML page (`golang.fyi/ch11/currency.html`) is responsible for displaying the result of a currency search. It uses JavaScript functions along with the `jQuery.js` library (not covered here) to post JSON-encoded requests to the backend Go service as shown in the following abbreviated HTML and JavaScript snippets:

```
<body>
<div class="container">
    <h2>Global Currency Service</h2>
    <p>Enter currency search string: <input id="in">
        <button type="button" class="btn btn-primary"
            onclick="doRequest()">Search</button>
    </p>
```

```
<table id="tbl" class="table table-striped">
  <thead>
    <tr>
      <th>Code</th>
      <th>Name</th>
      <th>Number</th>
      <th>Country</th>
    </tr>
  </thead>
  <tbody/>
</table>
</div>

<script>
var tbl = document.getElementById("tbl");
function addRow(code, name, number, country) {
  var rowCount = tbl.rows.length;
  var row = tbl.insertRow(rowCount);
  row.insertCell(0).innerHTML = code;
  row.insertCell(1).innerHTML = name;
  row.insertCell(2).innerHTML = number;
  row.insertCell(3).innerHTML = country;
}

function doRequest() {
param = document.getElementById("in").value
$.ajax('/currency', {
  method: 'PUT',
  contentType: 'application/json',
  processData: false,
  data: JSON.stringify({get:param})
}).then(
  function success(currencies) {
    currs = JSON.parse(currencies)
    for (i=0; i < currs.length; i++) {
      addRow(
        currs[i].currency_code,
        currs[i].currency_name,
        currs[i].currency_number,
        currs[i].currency_country
      );
    }
  });
}
</script>
```

golang.fyi/ch11/currency.html

A line-by-line analysis of the HTML and JavaScript code in this example is beyond the scope of the book; however, it is worth pointing out that the JavaScript `doRequest` function is where the interaction between the client and the server happens. It uses the jQuery's `$.ajax` function to build an HTTP request with a `PUT` method and to specify a JSON-encoded currency request object, `JSON.stringify({get: param})`, to send to the server. The `then` method accepts the callback function, `success(currencies)`, which handles the response from the server that parses displays in an HTML table.

When a search value is provided in the text box on the GUI, the page displays its results in the table dynamically as shown in the following screenshot:

Global Currency Service				
Enter currency search string: <input type="text" value="Franc"/> <button>Search</button>				
Code	Name	Number	Country	
XOF	CFA Franc BCEAO	952	BENIN	
XOF	CFA Franc BCEAO	952	BURKINA FASO	
BIF	Burundi Franc	108	BURUNDI	
XAF	CFA Franc BEAC	950	CAMEROON	
XAF	CFA Franc BEAC	950	CENTRAL AFRICAN REPUBLIC (THE)	
XAF	CFA Franc BEAC	950	CHAD	
KMF	Comoro Franc	174	COMOROS (THE)	

Summary

This chapter condenses several important notions about creating networked services in Go. It starts with a walkthrough of Go's `net` package including the `net.Conn` type to create a connection between network nodes, the `net.Dial` function to connect to a remote service, and the `net.Listen` function to handle incoming connections from a client. The chapter continues to cover different implementations of clients and server programs and shows the implications of creating custom protocols directly over raw TCP versus using an existing protocol such as HTTP with JSON data format.

The next chapter takes a different direction. It explores the packages, types, functions, and tools that are available in Go to facilitate source code testing.

12

Code Testing

Testing is a critical ritual of modern software development practices. Go brings testing directly into the development cycle by offering an API and command-line tool to seamlessly create and integrate automated test code. Here we will cover the Go testing suite, including the following:

- The Go test tool
- Writing Go tests
- HTTP testing
- Test coverage
- Code benchmark

The Go test tool

Prior to writing any test code, let's take a detour to discuss the tooling for automated testing in Go. Similar to the `go build` command, the `go test` command is designed to compile and exercise test source files in specified packages, as illustrated in the following command:

```
$> go test .
```

The previous command will exercise all test functions in the current package. Although it appears to be simple, the previous command accomplishes several complex steps, including:

- The compilation of all test files found in the current package
- Generating an instrumented binary from the test file
- Executing the test functions in the code

When the `go test` command targets multiple packages, the test tool generates multiple test binaries that are executed and tested independently, as shown in the following:

```
$> go test ./...
```

Test file names

The test command uses the import path standard (see [Chapter 6, Go Packages and Programs](#)) to specify which packages to test. Within a specified package, the test tool will compile all files with the `*_test.go` name pattern. For instance, assuming that we have a project that has a simple implementation of a mathematical vector type in a file called `vec.go`, a sensible name for its test file would be `vec_test.go`.

Test organization

Traditionally, test files are kept in the same package (directory) as the code being tested. This is because there is no need to separate tests files, as they are excluded from the compiled program binary. The following shows the directory layout for a typical Go package, in this instance the `fmt` package from the standard library. It shows all of the test files for the package in the same directory as the regular source code:

```
$>tree go/src/fmt/
├── doc.go
├── export_test.go
├── fmt_test.go
├── format.go
├── norace_test.go
├── print.go
├── race_test.go
├── scan.go
├── scan_test.go
└── stringer_test.go
```

Besides having a simpler project structure, keeping the files together gives test functions full visibility of the package being tested. This facilitates access to and verification of package elements that would otherwise be opaque to testing code. When your functions are placed in a separate package from the code to be tested, they lose access to non-exported elements of the code.

Writing Go tests

A Go test file is simply a set of functions with the following signature:

```
func Test<Name>(*testing.T)
```

Here, *<Name>* is an arbitrary name that reflects the purpose of the test. The test functions are intended to exercise a specific functional unit (or unit test) of the source code.

Before we write the test functions, let us review the code that will be tested. The following source snippet shows a simple implementation of a mathematical vector with Add, Sub, and Scale methods (see the full source code listed at <https://github.com/vladimirvivien/learning-go/ch12/vector/vec.go>). Notice that each method implements a specific behavior as a unit of functionality, which will make it easy to test:

```
type Vector interface {
    Add(other Vector) Vector
    Sub(other Vector) Vector
    Scale(factor float64)
    ...
}

func New(elems ...float64) SimpleVector {
    return SimpleVector(elems)
}

type SimpleVector []float64

func (v SimpleVector) Add(other Vector) Vector {
    v.assertLenMatch(other)
    otherVec := other.(SimpleVector)
    result := make([]float64, len(v))
    for i, val := range v {
        result[i] = val + otherVec[i]
    }
    return SimpleVector(result)
}
```

```
func (v SimpleVector) Sub(other Vector) Vector {
    v.assertLenMatch(other)
    otherVec := other.(SimpleVector)
    result := make([]float64, len(v))
    for i, val := range v {
        result[i] = val - otherVec[i]
    }
    return SimpleVector(result)
}

func (v SimpleVector) Scale(scale float64) {
    for i := range v {
        v[i] = v[i] * scale
    }
}
...
```

golang.fyi/ch12/vector/vec.go

The test functions

The test source code in file `vec_test.go` defines a series of functions that exercise the behavior of type `SimpleVector` (see the preceding section) by testing each of its methods independently:

```
import "testing"

func TestVectorAdd(t *testing.T) {
    v1 := New(8.218, -9.341)
    v2 := New(-1.129, 2.111)
    v3 := v1.Add(v2)
    expect := New(
        v1[0]+v2[0],
        v1[1]+v2[1],
    )

    if !v3.Eq(expect) {
        t.Logf("Addition failed, expecting %s, got %s",
            expect, v3)
        t.Fail()
    }
    t.Log(v1, "+", v2, v3)
}

func TestVectorSub(t *testing.T) {
    v1 := New(7.119, 8.215)
```

```
v2 := New(-8.223, 0.878)
v3 := v1.Sub(v2)
expect := New(
    v1[0]-v2[0],
    v1[1]-v2[1],
)
if !v3.Eq(expect) {
    t.Log("Subtraction failed, expecting %s, got %s",
        expect, v3)
    t.Fail()
}
t.Log(v1, "-", v2, "=", v3)
}

func TestVectorScale(t *testing.T) {
    v := New(1.671, -1.012, -0.318)
    v.Scale(7.41)
    expect := New(
        7.41*1.671,
        7.41*-1.012,
        7.41*-0.318,
    )
    if !v.Eq(expect) {
        t.Logf("Scalar mul failed, expecting %s, got %s",
            expect, v)
        t.Fail()
    }
    t.Log("1.671,-1.012, -0.318 Scale", 7.41, "=", v)
}
```

golang.fyi/ch12/vector/vec_test.go

As shown in the previous code, all test source code must import the "testing" package. This is because each test function receives an argument of type `*testing.T` as its parameter. As is discussed further in the chapter, this allows the test function to interact with the Go test runtime.

It is crucial to realize that each test function should be idempotent, with no reliance on any previously saved or shared states. In the previous source code snippet, each test function is executed as a standalone piece of code. Your test functions should not make any assumption about the order of execution as the Go test runtime makes no such guarantee.

The source code of a test function usually sets up an expected value, which is pre-determined based on knowledge of the tested code. That value is then compared to the calculated value returned by the code being tested. For instance, when adding two vectors, we can calculate the expected result using the rules of vector additions, as shown in the following snippet:

```
v1 := New(8.218, -9.341)
v2 := New(-1.129, 2.111)
v3 := v1.Add(v2)
expect := New(
    v1[0]+v2[0],
    v1[1]+v2[1],
)
```

In the preceding source snippet, the expected value is calculated using two simple vector values, `v1` and `v2`, and stored in the variable `expect`. Variable `v3`, on the other hand, stores the actual value of the vector, as calculated by the tested code. This allows us to test the actual versus the expected, as shown in the following:

```
if !v3.Eq(expect) {
    t.Log("Addition failed, expecting %s, got %s", expect, v3)
    t.Fail()
}
```

In the preceding source snippet, if the tested condition is `false`, then the test has failed. The code uses `t.Fail()` to signal the failure of the test function. Signaling failure is discussed in more detail in the Reporting failure section.

Running the tests

As mentioned in the introductory section of this chapter, test functions are executed using the `go test` command-line tool. For instance, if we run the following command from within the package `vector`, it will automatically run all of the test functions of that package:

```
$> cd vector
$> go test .
ok      github.com/vladimirvivien/learning-go/ch12/vector      0.001s
```

The test can also be executed by specifying a sub-package (or all packages with package wildcard `./...`) relative to where the command is issued, as shown in the following:

```
$> cd $GOPATH/src/github.com/vladimirvivien/learning-go/ch12/
$> go test ./vector
ok    github.com/vladimirvivien/learning-go/ch12/vector      0.005s
```

Filtering executed tests

During the development of a large set of test functions, it is often desirable to focus on a function (or set of functions) during debugging phases. The Go test command-line tool supports the `-run` flag, which specifies a regular expression that executes only functions whose names match the specified expression. The following command will only execute test function `TestVectorAdd`:

```
$> go test -run=VectorAdd -v
==== RUN  TestVectorAdd
--- PASS: TestVectorAdd (0.00s)
PASS
ok    github.com/vladimirvivien/learning-go/ch12/vector      0.025s
```

The use of the `-v` flag confirms that only one test function, `TestVectorAdd`, has been executed. As another example, the following executes all test functions that end with `VectorA.*$` or match function name `TestVectorMag`, while ignoring everything else:

```
> go test -run="VectorA.*$|TestVectorMag" -v
==== RUN  TestVectorAdd
--- PASS: TestVectorAdd (0.00s)
==== RUN  TestVectorMag
--- PASS: TestVectorMag (0.00s)
==== RUN  TestVectorAngle
--- PASS: TestVectorAngle (0.00s)
PASS
ok    github.com/vladimirvivien/learning-go/ch12/vector      0.043s
```

Test logging

When writing new or debugging existing test functions, it is often helpful to print information to a standard output. Type `testing.T` offers two logging methods: `Log`, which uses a default formatter, and `Logf`, which formats its output using formatting verbs (as defined in package `textfmt`). For instance, the following test function snippet from the `vector` example shows the code logging information with `t.Logf("Vector = %v; Unit vector = %v\n", v, expect)`:

```
func TestVectorUnit(t *testing.T) {
    v := New(5.581, -2.136)
    mag := v.Mag()
    expect := New((1/mag)*v[0], (1/mag)*v[1])
    if !v.Unit().Eq(expect) {
        t.Logf("Vector Unit failed, expecting %s, got %s",
            expect, v.Unit())
        t.Fail()
    }
    t.Logf("Vector = %v; Unit vector = %v\n", v, expect)
}
```

`golang.fyi/ch12/vector/vec_test.go`

As seen previously, the Go test tool runs tests with minimal output unless there is a test failure. However, the tool will output test logs when the verbose flag `-v` is provided. For instance, running the following in package `vector` will mute all logging statements:

```
> go test -run=VectorUnit
PASS
ok      github.com/vladimirvivien/learning-go/ch12/vector      0.005s
```

When the verbose flag `-v` is provided, as shown in the following command, the test runtime prints the output of the logs as shown:

```
$> go test -run=VectorUnit -v
==== RUN  TestVectorUnit
--- PASS: TestVectorUnit (0.00s)
vec_test.go:100: Vector = [5.581,-2.136]; Unit vector =
[0.9339352140866403,-0.35744232526233]
PASS
ok      github.com/vladimirvivien/learning-go/ch12/vector      0.001s
```

Reporting failure

By default, the Go test runtime considers a test a success if the test function runs and returns normally without a panic. For example, the following test function is broken, since its expected value is not properly calculated. The test runtime, however, will always report it as passing because it does not include any code to report the failure:

```
func TestVectorDotProd(t *testing.T) {
    v1 := New(7.887, 4.138).(*SimpleVector)
    v2 := New(-8.802, 6.776).(*SimpleVector)
    actual := v1.DotProd(v2)
    expect := v1[0]*v2[0] - v1[1]*v2[1]
    if actual != expect {
        t.Logf("DotProduct failed, expecting %d, got %d",
            expect, actual)
    }
}
```

golang.fyi/ch12/vec_test.go

This false positive condition may go unnoticed, especially if the verbose flag is turned off, minimizing any visual clues that it is broken:

```
$> go test -run=VectorDot
PASS
ok      github.com/vladimirvivien/learning-go/ch12/vector      0.001s
```

One way the previous test can be fixed is by using the `Fail` method from type `testing.T` to signal failure, as shown in the following snippet:

```
func TestVectorDotProd(t *testing.T) {
    ...
    if actual != expect {
        t.Logf("DotProduct failed, expecting %d, got %d",
            expect, actual)
        t.Fail()
    }
}
```

So now, when the test is executed, it correctly reports that it is broken, as shown in the following output:

```
$> go test -run=VectorDot
--- FAIL: TestVectorDotProd (0.00s)
vec_test.go:109: DotPoduct failed, expecting -97.460462, got -41.382286
FAIL
exit status 1
FAIL  github.com/vladimirvivien/learning-go/ch12/vector      0.002s
```

It is important to understand that method `Fail` only reports failure and does not halt the execution of a test function. On the other hand, when it makes sense to actually exit the function upon a failed condition, the test API offers the method `FailNow`, which signals failure and exits the currently executing test function.

Type `testing.T` provides the convenience methods `Logf` and `Errorf`, which combine both logging and failure reporting. For instance, the following snippet uses the `Errorf` method, which is equivalent to calling the `Logf` and `Fail` methods:

```
func TestVectorMag(t *testing.T) {
    v := New(-0.221, 7.437)
    expected := math.Sqrt(v[0]*v[0] + v[1]*v[1])
    if v.Mag() != expected {
        t.Errorf("Magnitude failed, execpted %d, got %d",
            expected, v.Mag())
    }
}
```

golang.fyi/ch12/vector/vec.go

Type `testing.T` also offers `Fatal` and `Formatf` methods as a way of combining the logging of a message and the immediate termination of a test function.

Skipping tests

It is sometimes necessary to skip test functions due to a number of factors such as environment constraints, resource availability, or inappropriate environment settings. The testing API makes it possible to skip a test function using the `SkipNow` method from type `testing.T`. The following source code snippet will only run the test function when the arbitrary operating system environment variable named `RUN_ANGLE` is set. Otherwise, it will skip the test:

```
func TestVectorAngle(t *testing.T) {
    if os.Getenv("RUN_ANGLE") == "" {
```

```
        t.Skipf("Env variable RUN_ANGLE not set, skipping:")
    }
    v1 := New(3.183, -7.627)
    v2 := New(-2.668, 5.319)
    actual := v1.Angle(v2)
    expect := math.Acos(v1.DotProd(v2) / (v1.Mag() * v2.Mag()))
    if actual != expect {
        t.Logf("Vector angle failed, expecting %d, got %d",
            expect, actual)
        t.Fail()
    }
    t.Log("Angle between", v1, "and", v2, "=", actual)
}
```

Notice the code is using the `Skipf` method, which is a combination of the methods `SkipNow` and `Logf` from type `testing.T`. When the test is executed without the environment variable, it outputs the following:

```
$> go test -run=Angle -v
==> RUN      TestVectorAngle
--- SKIP: TestVectorAngle (0.00s)
        vec_test.go:128: Env variable RUN_ANGLE not set, skipping:
PASS
ok      github.com/vladimirvivien/learning-go/ch12/vector      0.006s
```

When the environment variable is provided, as is done with the following Linux/Unix command, the test executes as expected (consult your OS on how to set environment variables):

```
> RUN_ANGLE=1 go test -run=Angle -v
==> RUN      TestVectorAngle
--- PASS: TestVectorAngle (0.00s)
        vec_test.go:138: Angle between [3.183,-7.627] and [-2.668,5.319]
= 3.0720263098372476
PASS
ok      github.com/vladimirvivien/learning-go/ch12/vector      0.005s
```

Table-driven tests

One technique you often encounter in Go is the use of table-driven tests. This is where a set of input and expected output is stored in a data structure, which is then used to cycle through different test scenarios. For instance, in the following test function, the `cases` variable, of type `[]struct{vec SimpleVector; expected float64}`, to store several vector values and their expected magnitude values used to test the vector method `Mag`:

```
func TestVectorMag(t *testing.T) {
    cases := []struct{
        vec SimpleVector
        expected float64
    }{
        {New(1.2, 3.4), math.Sqrt(1.2*1.2 + 3.4*3.4)},
        {New(-0.21, 7.47), math.Sqrt(-0.21*-0.21 + 7.47*7.47)},
        {New(1.43, -5.40), math.Sqrt(1.43*1.43 + -5.40*-5.40)},
        {New(-2.07, -9.0), math.Sqrt(-2.07*-2.07 + -9.0*-9.0)},
    }
    for _, c := range cases {
        mag := c.vec.Mag()
        if mag != c.expected {
            t.Errorf("Magnitude failed, execpted %d, got %d",
                    c.expected, mag)
        }
    }
}
```

golang.fyi/ch12/vector/vec.go

With each iteration of the loop, the code tests the value calculated by the `Mag` method against an expected value. Using this approach, we can test several combinations of input and their respective output, as is done in the preceding code. This technique can be expanded as necessary to include more parameters. For instance, a name field can be used to name each case, which is useful when the number of test cases is large. Or, to be even more fancy, one can include a function field in the test case struct to specify custom logic to use for each respective case.

HTTP testing

In Chapter 11, *Writing Networked Services*, we saw that Go offers first-class APIs to build client and server programs using HTTP. The `net/http/httptest` sub-package, part of the Go standard library, facilitates the testing automation of both HTTP server and client code, as discussed in this section.

To explore this space, we will implement a simple API service that exposes the vector operations (covered in earlier sections) as HTTP endpoints. For instance, the following source snippet partially shows the methods that make up the server (for a complete listing, see <https://github.com/vladimirvivien/learning-go/ch12/service/serv.go>):

```
package main

import (
    "encoding/json"
    "fmt"
    "net/http"

    "github.com/vladimirvivien/learning-go/ch12/vector"
)
func add(resp http.ResponseWriter, req *http.Request) {
    var params []vector.SimpleVector
    if err := json.NewDecoder(req.Body).Decode(&params);
        err != nil {
        resp.WriteHeader(http.StatusBadRequest)
        fmt.Fprintf(resp, "Unable to parse request: %s\n", err)
        return
    }
    if len(params) != 2 {
        resp.WriteHeader(http.StatusBadRequest)
        fmt.Fprintf(resp, "Expected 2 or more vectors")
        return
    }
    result := params[0].Add(params[1])
    if err := json.NewEncoder(resp).Encode(&result); err != nil {
        resp.WriteHeader(http.StatusInternalServerError)
        fmt.Fprintf(resp, err.Error())
        return
    }
}
...
func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/vec/add", add)
    mux.HandleFunc("/vec/sub", sub)
```

```
    mux.HandleFunc("/vec/dotprod", dotProd)
    mux.HandleFunc("/vec/mag", mag)
    mux.HandleFunc("/vec/unit", unit)

    if err := http.ListenAndServe(":4040", mux); err != nil {
        fmt.Println(err)
    }
}
```

golang.fyi/ch12/service/serv.go

Each function (add, sub, dotprod, mag, and unit) implements the `http.Handler` interface. The functions are used to handle HTTP requests from the client to calculate the respective operations from the `vector` package. Both requests and responses are formatted using JSON for simplicity.

Testing HTTP server code

When writing HTTP server code, you will undoubtedly run into the need to test your code, in a robust and repeatable manner, without having to set up some fragile code harness to simulate end-to-end testing. Type `httptest.ResponseRecorder` is designed specifically to provide unit testing capabilities for exercising the HTTP handler methods by inspecting state changes to the `http.ResponseWriter` in the tested function. For instance, the following snippet uses `httptest.ResponseRecorder` to test the server's add method:

```
import (
    "net/http"
    "net/http/httptest"
    "strconv"
    "strings"
    "testing"

    "github.com/vladimirvivien/learning-go/ch12/vector"
)

func TestVectorAdd(t *testing.T) {
    reqBody := "[[1,2],[3,4]]"
    req, err := http.NewRequest(
        "POST", "http://0.0.0.0/", strings.NewReader(reqBody))
    if err != nil {
        t.Fatal(err)
    }
    actual := vector.New(1, 2).Add(vector.New(3, 4))
    w := httptest.NewRecorder()
    add(w, req)
```

```
        if actual.String() != strings.TrimSpace(w.Body.String()) {
            t.Fatalf("Expecting actual %s, got %s",
                    actual.String(), w.Body.String(),
            )
        }
    }
```

The code uses `reg, err := http.NewRequest("POST", "http://0.0.0.0/", strings.NewReader(reqBody))` to create a new `*http.Request` value with a "POST" method, a fake URL, and a request body, variable `reqBody`, encoded as a JSON array. Later in the code, `w := httptest.NewRecorder()` is used to create an `httputil.ResponseRecorder` value, which is used to invoke the `add(w, req)` function along with the created request. The value recorded in `w`, during the execution of function `add`, is compared with expected value stored in `actual` with `if actual.String() != strings.TrimSpace(w.Body.String()){...}`.

Testing HTTP client code

Creating test code for an HTTP client is more involved, since you actually need a server running for proper testing. Luckily, package `httptest` provides type `httptest.Server` to programmatically create servers to test client requests and send back mock responses to the client.

To illustrate, let us consider the following code, which partially shows the implementation of an HTTP client to the vector server presented earlier (see the full code listing at <https://github.com/vladimirvivien/learning-go/ch12/client/client.go>). The `add` method encodes the parameters `vec0` and `vec2` of type `vector.SimpleVector` as JSON objects, which are sent to the server using `c.client.Do(req)`. The response is decoded from the JSON array into type `vector.SimpleVector` assigned to variable `result`:

```
type vecClient struct {
    svcAddr string
    client *http.Client
}
func (c *vecClient) add(
    vec0, vec1 vector.SimpleVector) (vector.SimpleVector, error) {
    uri := c.svcAddr + "/vec/add"

    // encode params
    var body bytes.Buffer
    params := []vector.SimpleVector{vec0, vec1}
    if err := json.NewEncoder(&body).Encode(&params); err != nil {


```

```
        return []float64{}, err
    }
    req, err := http.NewRequest("POST", uri, &body)
    if err != nil {
        return []float64{}, err
    }

    // send request
    resp, err := c.client.Do(req)
    if err != nil {
        return []float64{}, err
    }
    defer resp.Body.Close()

    // handle response
    var result vector.SimpleVector
    if err := json.NewDecoder(resp.Body).
        Decode(&result); err != nil {
        return []float64{}, err
    }
    return result, nil
}
```

golang.fyi/ch12/client/client.go

We can use type `httptest.Server` to create code to test the requests sent by a client and to return data to the client code for further inspection. Function `httptest.NewServer` takes a value of type `http.Handler`, where the test logic for the server is encapsulated. The function then returns a new running HTTP server ready to serve on a system-selected port.

The following test function shows how to use `httptest.Server` to exercise the `add` method from the client code presented earlier. Notice that when creating the server, the code uses type `http.HandlerFunc`, which is an adapter that takes a function value to produce an `http.Handler`. This convenience allows us to skip the creation of a separate type to implement a new `http.Handler`:

```
import (
    "net/http"
    "net/http/httptest"
    ...
)
func TestClientAdd(t *testing.T) {
    server := httptest.NewServer(http.HandlerFunc(
        func(resp http.ResponseWriter, req *http.Request) {
            // test incoming request path
            if req.URL.Path != "/vec/add" {

```

```
        t.Errorf("unexpected request path %s",
                  req.URL.Path)
        return
    }
    // test incoming params
    body, _ := ioutil.ReadAll(req.Body)
    params := strings.TrimSpace(string(body))
    if params != "[[1,2],[3,4]]" {
        t.Errorf("unexpected params '%v'", params)
        return
    }
    // send result
    result := vector.New(1, 2).Add(vector.New(3, 4))
    err := json.NewEncoder(resp).Encode(&result)
    if err != nil {
        t.Fatal(err)
        return
    }
},
))
defer server.Close()
client := newVecClient(server.URL)
expected := vector.New(1, 2).Add(vector.New(3, 4))
result, err := client.add(vector.New(1, 2), vector.New(3, 4))
if err != nil {
    t.Fatal(err)
}
if !result.Eq(expected) {
    t.Errorf("Expecting %s, got %s", expected, result)
}
}
```

golang.fyi/ch12/client/client_test.go

The test function first sets up the server along with its handler function. Inside the function of `http.HandlerFunc`, the code first ensures that the client requests the proper path of `"/vec/add"`. Next, the code inspects the request body from the client, ensuring proper JSON format and valid parameters for the add operation. Finally, the handler function encodes the expected result as JSON and sends it as a response to the client.

The code uses the system-generated `server` address to create a new `client` with `newVecClient(server.URL)`. Method call `client.add(vector.New(1, 2), vector.New(3, 4))` sends a request to the test server to calculate the vector addition of the two values in its parameter list. As shown earlier, the test server merely simulates the real server code and returns the calculated vector value. The `result` is inspected against the `expected` value to ensure proper working of the `add` method.

Test coverage

When writing tests, it is often important to know how much of the actual code is getting exercised (or covered) by the tests. That number is an indication of the penetration of the test logic against the source code. Whether you agree or not, in many software development practices, test coverage is a critical metric as it is a measure of how well the code is tested.

Fortunately, the Go test tool comes with a built-in coverage tool. Running the Go test command with the `-cover` flag instruments the original source code with coverage logic. It then runs the generated test binary, providing a summary of the overall coverage profile of the package, as shown in the following:

```
$> go test -cover
PASS
coverage: 87.8% of statements
ok    github.com/vladimirvivien/learning-go/ch12/vector      0.028s
```

The result shows a well-tested code with a coverage number of 87.8%. We can use the test tool to extract more details about the section of the code that is tested. To do this, we use the `-coverprofile` flag to record coverage metrics to a file, as shown:

```
$> go test -coverprofile=cover.out
```

The cover tool

Once the coverage data is saved, it can be presented in a textual tab-formatted table using the `go tool cover` command. The following shows a partial output of the breakdown of the coverage metrics for each tested function in the coverage file generated previously:

```
$> go tool cover -func=cover.out
...
learning-go/ch12/vector/vec.go:52: Eq      100.0%
learning-go/ch12/vector/vec.go:57: Eq2     83.3%
learning-go/ch12/vector/vec.go:74: Add     100.0%
learning-go/ch12/vector/vec.go:85: Sub     100.0%
learning-go/ch12/vector/vec.go:96: Scale    100.0%
...
```

The `cover` tool can overlay the coverage metrics over the actual code, providing a visual aid to show the covered (and uncovered) portion of the code. Use the `-html` flag to generate an HTML page using the coverage data gathered previously:

```
$> go tool cover -html=cover.out
```

The command opens the installed default web browser and displays the coverage data, as shown in the following screenshot:

```
// Eq compares vector magnitude and directions
func (v SimpleVector) Eq(other Vector) bool {
    return v.Mag() == other.Mag() && (v.Angle(other) <= zero || math.I...
}

// Eq compares each vector components for equality
func (v SimpleVector) Eq2(other Vector) bool {
    v.assertLenMatch(other)
    otherVec := other.(SimpleVector)
    for i, val := range v {
        if val != otherVec[i] {
            return false
        }
    }
    return true
}

// Test for the zero vector
func (v SimpleVector) IsZero() bool {
    return v.Mag() <= zero
}

// Add returns the sum of two vectors
func (v SimpleVector) Add(other Vector) Vector {
    v.assertLenMatch(other)
    otherVec := other.(SimpleVector)
    result := make([]float64, len(v))
    for i, val := range v {
        result[i] = val + otherVec[i]
    }
    return SimpleVector(result)
}
```

The preceding screenshot shows only a portion of the generated HTML page. It shows covered code in green and code that is not covered in red. Anything else is displayed in gray.

Code benchmark

The purpose of benchmarking is to measure a code's performance. The Go test command-line tool comes with support for the automated generation and measurement of benchmark metrics. Similar to unit tests, the test tool uses benchmark functions to specify what portion of the code to measure. The benchmark function uses the following function naming pattern and signature:

```
func Benchmark<Name>(*testing.B)
```

Benchmark functions are expected to have names that start with *benchmark* and accept a pointer value of type `*testing.B`. The following shows a function that benchmarks the `Add` method for type `SimpleVector` (introduced earlier):

```
import (
    "math/rand"
    "testing"
    "time"
)

...
func BenchmarkVectorAdd(b *testing.B) {
    r := rand.New(rand.NewSource(time.Now().UnixNano()))
    for i := 0; i < b.N; i++ {
        v1 := New(r.Float64(), r.Float64())
        v2 := New(r.Float64(), r.Float64())
        v1.Add(v2)
    }
}
```

golang.fyi/ch12/vector/vec_bench_test.go

Go's test runtime invokes the benchmark functions by injecting pointer `*testing.B` as a parameter. That value defines methods for interacting with the benchmark framework such as logging, failure-signaling, and other functionalities similar to type `testing.T`. Type `testing.B` also offers additional benchmark-specific elements, including an integer field `N`. It is intended to be the number of iterations that the benchmark function should use for effective measurements.

The code being benchmarked should be placed within a `for` loop bounded by `N`, as illustrated in the previous example. For the benchmark to be effective, there should be no variances in the size of the input for each iteration of the loop. For instance, in the preceding benchmark, each iteration always uses a vector of size 2 (while the actual values of the vectors are randomized).

Running the benchmark

Benchmark functions are not executed unless the test command-line tool receives the flag `-bench`. The following command runs all the benchmarks functions in the current package:

```
$> go test -bench=.  
PASS  
BenchmarkVectorAdd-2      2000000      761 ns/op  
BenchmarkVectorSub-2      2000000      788 ns/op  
BenchmarkVectorScale-2    5000000      269 ns/op  
BenchmarkVectorMag-2     5000000      243 ns/op  
BenchmarkVectorUnit-2    3000000      507 ns/op  
BenchmarkVectorDotProd-2  3000000      549 ns/op  
BenchmarkVectorAngle-2   2000000      659 ns/op  
ok   github.com/vladimirvivien/learning-go/ch12/vector    14.123s
```

Before dissecting the benchmark result, let us understand the previously issued command. The `go test -bench=.` command first executes all the test functions in the package followed by all the benchmark functions (you can verify this by adding the verbose flag `-v` to the command).

Similar to the `-run` flag, the `-bench` flag specifies a regular expression used to select the benchmark functions that get executed. The `-bench=.` flag matches the name of all benchmark functions, as shown in the previous example. The following, however, only runs benchmark functions that contain the pattern "VectorA" in their names. This includes the `BenchmarkVectorAngle()` and `BenchmarkVectorAngle()` functions:

```
$> go test -bench="VectorA"  
PASS  
BenchmarkVectorAdd-2      2000000      764 ns/op  
BenchmarkVectorAngle-2    2000000      665 ns/op  
ok   github.com/vladimirvivien/learning-go/ch12/vector    4.396s
```

Skipping test functions

As mentioned previously, when benchmarks are executed, the test tool will also run all test functions. This may be undesirable, especially if you have a large number of tests in your package. A simple way to skip the test functions during benchmark execution is to set the `-run` flag to a value that matches no test functions, as shown in the following:

```
> go test -bench=. -run=NONE -v  
PASS  
BenchmarkVectorAdd-2      2000000      791 ns/op  
BenchmarkVectorSub-2      2000000      777 ns/op
```

```
...
BenchmarkVectorAngle-2      2000000      653 ns/op
ok   github.com/vladimirvivien/learning-go/ch12/vector    14.069s
```

The previous command only executes benchmark functions, as shown by the partial verbose output. The value of the `-run` flag is completely arbitrary and can be set to any value that will cause it to skip the execution of test functions.

The benchmark report

Unlike tests, a benchmark report is always verbose and displays several columns of metrics, as shown in the following:

```
$> go test -run=None -bench="Add|Sub|Scale"
PASS
BenchmarkVectorAdd-2      2000000      800 ns/op
BenchmarkVectorSub-2      2000000      798 ns/op
BenchmarkVectorScale-2    5000000      266 ns/op
ok   github.com/vladimirvivien/learning-go/ch12/vector    6.473s
```

The first column contains the names of the benchmark functions, with each name suffixed with a number that reflects the value of `GOMAXPROCS`, which can be set at test time using the `-cpu` flag (relevant for running benchmarks in parallel).

The next column displays the number of iterations for each benchmark loop. For instance, in the previous report, the first two benchmark functions looped 2 million times, while the final benchmark function iterated 5 million times. The last column of the report shows the average time it takes to execute the tested function. For instance, the 5 million calls to the `Scale` method executed in benchmark function `BenchmarkVectorScale` took on average 266 nanoseconds to complete.

Adjusting N

By default, the test framework gradually adjusts `N` to be large enough to arrive at stable and meaningful metrics over a period of *one second*. You cannot change `N` directly. However, you can use flag `-benctime` to specify a benchmark run time and thus influence the number of iterations during a benchmark. For instance, the following runs the benchmark for a period of 5 seconds:

```
> go test -run=Bench -bench="Add|Sub|Scale" -benctime 5s
PASS
BenchmarkVectorAdd-2      10000000      784 ns/op
```

```
BenchmarkVectorSub-2      10000000          810 ns/op
BenchmarkVectorScale-2    30000000          265 ns/op
ok   github.com/vladimirvivien/learning-go/ch12/vector      25.877s
```

Notice that even though there is a drastic jump in the number iterations (factor of five or more) for each benchmark, the average performance time for each benchmark function remains reasonably consistent. This information provides valuable insight into the performance of your code. It is a great way to observe the impact of code or load changes on performance, as discussed in the following section.

Comparative benchmarks

Another useful aspect of benchmarking code is to compare the performance of different algorithms that implement similar functionalities. Exercising the algorithms using performance benchmarks will indicate which of the implementations may be more compute- and memory-efficient.

For instance, two vectors are said to be equal if they have the same magnitude and same direction (or have an angle value of zero between them). We can implement this definition using the following source snippet:

```
const zero = 1.0e-7
...
func (v SimpleVector) Eq(other Vector) bool {
    ang := v.Angle(other)
    if math.IsNaN(ang) {
        return v.Mag() == other.Mag()
    }
    return v.Mag() == other.Mag() && ang <= zero
}
```

golang.fyi/ch12/vector/vec.go

When the preceding method is benchmarked, it yields to the following result. Each of its 3 million iterations takes an average of half a millisecond to run:

```
$> go test -run=Bench -bench=Equal1
PASS
BenchmarkVectorEqual1-2  3000000          454 ns/op
ok   github.com/vladimirvivien/learning-go/ch12/vector      1.849s
```

The benchmark result is not bad, especially when compared to the other benchmarked methods that we saw earlier. However, suppose we want to improve on the performance of the `Eq` method (maybe because it is a critical part of a program). We can use the `-benchmem` flag to get additional information about the benchmarked test:

```
$> go test -run=bench -bench=Equal1 -benchmem
PASS
BenchmarkVectorEqual1-2 3000000 474 ns/op 48 B/op 2 allocs/op
```

The `-benchmem` flag causes the test tool to reveal two additional columns, which provide memory allocation metrics, as shown in the previous output. We see that the `Eq` method allocates a total of 48 bytes, with two allocations calls per operation.

This does not tell us much until we have something else to compare it to. Fortunately, there is another equality algorithm that we can try. It is based on the fact that two vectors are also equal if they have the same number of elements and each element is equal. This definition can be implemented by traversing the vector and comparing its elements, as is done in the following code:

```
func (v SimpleVector) Eq2(other Vector) bool {
    v.assertLenMatch(other)
    otherVec := other.(SimpleVector)
    for i, val := range v {
        if val != otherVec[i] {
            return false
        }
    }
    return true
}
```

golang.fyi/ch12/vector/vec.go

Now let us benchmark the `Eq` and `Eq2` equality methods to see which is more performant, as done in the following:

```
$> go test -run=bench -bench=Equal -benchmem
PASS
BenchmarkVectorEqual1-2 3000000 447 ns/op 48 B/op 2 allocs/op
BenchmarkVectorEqual2-2 5000000 265 ns/op 32 B/op 1 allocs/op
```

According to the benchmark report, method `Eq2` is more performant of the two equality methods. It runs in about half the time of the original method, with considerably less memory allocated. Since both benchmarks run with similar input data, we can confidently say the second method is a better choice than the first.



Depending on Go version and machine size and architecture, these benchmark numbers will vary. However, the result will always show that the Eq2 method is more performant.

This discussion only scratches the surface of comparative benchmarks. For instance, the previous benchmark tests use the same size input. Sometimes it is useful to observe the change in performance as the input size changes. We could have compared the performance profile of the equality method as we change the size of the input, say, from 3, 10, 20, or 30 elements. If the algorithm is sensitive size, expanding the benchmark using such attributes will reveal any bottlenecks.

Summary

This chapter provided a broad introduction to the practice of writing tests in Go. It discussed several key topics, including the use of the `go test` tool to compile and execute automated tests. Readers learned how to write test functions to ensure their code is properly tested and covered. The chapter also discussed the topic of testing HTTP clients and servers. Finally, the chapter introduced the topic of benchmarking as a way to automate, analyze, and measure code performance using built-in Go tools.

Index

A

address operator 89
anonymous field, struct
 about 175
 promoted fields 176
arithmetic operators 49
array
 about 148, 149
 array traversal 153
 as parameters 154, 155
 initialization 149, 150, 151
 length and capacity 153
 named array types, declaring 151
 traversal 154
 using 152
assignment operators 50
attributes, slice
 a capacity 158
 a length 158
 a pointer 158

B

bitwise operators 50
blank identifier 139
 about 32
 package imports, muting 32
 unwanted function results, muting 33
Boolean type 84
break statement 73, 74
buffered IO
 about 249
 buffer, scanning 251
 buffered writers and readers 250
bufio package
 reference 249
built-in identifiers

about 33
functions 34
types 33
values 34

C

code benchmark
 about 312
 benchmark report 314
 comparative benchmarks 315, 316
 N, adjusting 314
 running 313
 test functions, skipping 313
code coverage 22
code testing
 about 293
comparison operators 51
complex number types
 complex128 82
 complex64 82
composite types
 about 17, 148
 array 148
 map 167
 slice 155
 struct 172
constant enumeration
 about 46
 default enumeration type, overriding 47
 enumerated values, skipping 48
 iota, using in expressions 47
constants
 about 42
 declaration block 45
 literals 42
 typed constants 43
 untyped constants 43

continue statement 73, 74, 75
cURL command line tool
 reference 286
curr package
 reference 269
curr1 package
 reference 284

D

data
 binary encoding, with gob 254
 custom decoding 260
 custom encoding 260
 decoding 253
 encoding 253
 encoding, as JSON 256
decrement operators 49
default ServeMux 283
documentation
 about 23
 URL 23
dot identifier 139

E

encoding package
 reference 253
error
 error type 114
 example, reference link 111
 handling 110, 114
 signaling 111
 signalling 110, 113
Euclidian division algorithm
 reference link 103
expressions
 iota, using 47
extensive library 24

F

files
 creating 241
 opening 242
 reading 243
 standard error 245
 standard input 245

standard output 245
working with 241
writing 243
floating point types
 float64 82
 float 32 82
fmt
 reference 246
for statements
 about 66
 for condition 66
 infinite loop 67
 range 70, 71
 traditional for statement 68, 69
formatted IO, with fmt
 about 246
 io.Reader, reading from 247
 io.Writer interfaces, printing to 246
 standard input, reading from 249
 standard output, printing to 247
function calls
 defer, using 117
 deferring 115
function result parameters
 about 103
 function result parameters 105
 named result parameters 104
functions
 about 15
 panic recovery 119, 120, 121
 panicking 117, 118

G

Go functions
 about 97, 99
 declaration 98
 function type 101, 102
 result parameters 103
 signature 101
 variadic parameters 102, 103
Go package
 about 122, 123, 124
 import path 127
 workspace 124, 125
Go Playground

about 10
IDE, avoiding 11
URL 10

Go test tool
about 293
test file names 294
test organization 294

Go tests
executed tests, filtering 299
failure, reporting 301
running 298
skipping 302
table-driven tests 304
test functions 296, 298
test logging 300
writing 295

Go Toolchain
about 25
URL 12

Go types
about 77
array 79
chan T 80
func (T) R 80
interface{} 79
map[K]T 79
slice 79
struct {} 79

Go
about 8, 9
channels 20
compilation 22
composite types 17
concurrency 20
design 13
functions 15
installing 11
interfaces 19
memory management 21
methods 18
named types 18
objects 18
packages 16
safety 21
source code examples 12

statically typed values 16
URL, for installing 11
workspace 16

gob package
reference 254

gob
about 254

goroutine 20

goto statement 73, 75

H

Hello World program
writing 12

higher-order functions
about 109

HTTP package
about 275
client requests, handling 278
http.Client type 275, 277
http.Client type, configuring 277, 278
requests, routing with http.ServeMux 282
responses, handling 279
simple HTTP server 279, 280, 281

HTTP testing
HTTP client code, testing 307, 309
HTTP server code, testing 306

http.Client struct 275

I

IDE plugins
URL 11

identifiers
about 32
attributes 32
blank identifier 32
built-in identifiers 33

if statement
about 53
initialization 57

import path 127

in-memory IO 252

increment operators 50

io package
reference 239
working with 239

`io.Reader` interface
 about 233, 234
 readers, chaining 234, 236
`io.Writer` interface
 about 236, 238
`IO`
 with, readers and writers 233
`iota`
 using, in expressions 47

J

`JSON API` server
 about 284, 286
 API server client, in Go 287
 JavaScript API server client 288, 291
 testing, with cURL 286
`JSON` mapping
 controlling, with struct tags 259

L

`label` identifier 73
`logical operators` 51

M

`map` functions
 `delete(map,key)` 171
 `len(map)` 171
`map`
 about 167
 as parameter 171
 creating 168
 functions 171
 initialization 167, 168
 traversal 170
 using 169
`methods` 18
`multi-file packages` 130

N

`named types` 18
`naming, packages`
 about 131
 context, adding to path 132
 short names, using 132

`unique namespace, using` 131
`net` package
 about 264
 addressing 265
 client connections, accepting 268
 connection, dialing 265
 incoming connections, listening for 267
 `net.Conn` Type 265
 reference 264
`net/http` package
 reference 275
`networked services`
 writing 264
`new()` function 91
`numeric types`
 about 80
 complex number types 82
 floating point types 82
 numeric literals 82
 signed integer types 81
 unsigned integer types 81

O

`objects, Go` 18
`operators`
 about 49
 arithmetic operators 49
 assignment operators 50
 bitwise operators 50
 comparison operators 51
 decrement operators 49
 increment operators 49
 logical operators 51
 precedence 52
`os` package
 reference 241
`os.OpenFile` function
 about 242, 243

P

`package identifier`
 specifying 138
`package visibility`
 about 134, 135
`package member visibility` 135

packages
about 16
blank identifier 139
building 133
creating 128, 129
declaring 129
dot identifier 139
identifier, specifying 138
importing 136
initialization 140
installing 134
multi-file packages 130
naming 131

parameter values
anonymous function literals, invoking 108
closures 108
pass-by-reference, achieving 106, 107
passing 105, 106

pointer type 89

pointers
about 88
address operator 89
ne() function 91
pointer type 89
referenced values, accessing 92

profiling 22

programs
arguments, accessing 143, 144
building 145
creating 141, 143
installing 145

R

remote packages
about 146

remote procedure calls (RPC) 253

reusable errors
declaration, reference link 115

rules 131

rune type 84, 85, 86

S

short variable declaration
about 39
restrictions 40

signed integer types
int 82
int16 81
int32 81
int64 81
int8 81

simple HTTP server
default server 281

slice of bytes ([] byte) 233

slice
about 155
appending, to slices 165
array, slicing 161
creating, by slicing array 159
existing slice, slicing 160
expression, examples 159
initialization 157
length and capacity 164
representation 158
slice expressions, with capacity 161
slice, creating 162
slices, copying 165
slices, using 163
strings, using as slices 166
using, as parameters 164

source file
about 27
multiple lines 30
optional semicolon 29

string type
about 84, 86
interpreted and raw string literals 87

struct tags
JSON mapping, controlling with 259

struct
about 172
anonymous field 175
as parameters 177, 178
field tags 178
fields, accessing 173
initialization 173, 174
named struct types, declaring 174

switch statements
about 57
expression switches, using 59

expressionless switches 62
fallthrough cases 61
initializer 63, 64
type switches 64

T

TCP API server
about 269, 270, 272
connecting to, with Go 273
connecting to, with telnet 272
test coverage
about 310
cover tool 310, 311
testing 22
type
conversion, scenarios 96
converting 94, 96
declaring 93
typed constants 43

U

unsigned integer types
byte 81
uint 81
uint16 81
uint32 81

uint64 81
uint8 81
uintptr 81
untyped constants
about 43
assigning 44

V

variables
about 34
declaration 34
declaration block 42
initialized declaration 36
scope 40
short variable declaration 39
types, omitting 37
visibility 40
zero-value 36
variadic (variable length arguments) 102

W

workspace, Go package
about 124, 125
bin directory 125
creating 126, 127
pkg directory 126
src directory 126

Module 2

Go Design Patterns

Learn idiomatic, efficient, clean, and extensible Go design and concurrency patterns by using TDD

1

Ready... Steady... Go!

Design Patterns have been the foundation for hundreds of thousands of pieces of software. Since the *Gang Of Four* (Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides) wrote the book *Design Patterns: Elements of Reusable Object-Oriented Software* in 1994 with examples in C++ and Smalltalk, the twenty-three classic patterns have been re-implemented in most of major languages of today and they have been used in almost every project you know about.

The *Gang of Four* detected that many small architectures were present in many of their projects, they started to rewrite them in a more abstract way and they released the famous book.

This book is a comprehensive explanation and implementation of the most common design patterns from the *Gang of Four* and today's patterns plus some of the most idiomatic concurrency patterns in Go.

But what is Go...?

A little bit of history

On the last 20 years, we have lived an incredible growth in computer science. Storage spaces have been increased dramatically, RAM has suffered a substantial growth, and CPU's are... well... simply faster. Have they grown as much as storage and RAM memory? Not really, CPU industry has reached a limit in the speed that their CPU's can deliver, mainly because they have become so fast that they cannot get enough power to work while they dissipate enough heat. The CPU manufacturers are now shipping more cores on each computer. This situation crashes against the background of many systems programming languages that weren't designed for multi-processor CPUs or large distributed systems that act as a unique machine. In Google, they realized that this was becoming more than an issue while they were struggling to develop distributed applications in languages like Java or C++ that weren't designed with concurrency in mind.

At the same time, our programs were bigger, more complex, more difficult to maintain and with a lot of room for bad practices. While our computers had more cores and were faster, we were not faster when developing our code neither our distributed applications. This was Go's target.

Go design started in 2007 by three Googlers in the research of a programming language that could solve common issues in large scale distributed systems like the ones you can find at Google. The creators were:

- Rob Pike: Plan 9 and Inferno OS.
- Robert Griesemer: Worked at Google's V8 JavaScript engine that powers Google Chrome.
- Ken Thompson: Worked at Bell labs and the Unix team. It has been involved in designing of the Plan 9 operating system as well as the definition of the UTF-8 encoding.

In 2008, the compiler was done and the team got the help of Russ Cox and Ian Lance Taylor. The team started their journey to open source the project in 2009 and in March 2012 they reached a version 1.0 after more than fifty releases.

Installing Go

Any Go Installation needs two basic things: the binaries of the language somewhere on your disk and a **GOPATH** path in your system where your projects and the projects that you download from other people will be stored.

In the following lines, we will explore how to install Go binaries in Linux, Windows and OS X. For a detailed explanation of how to install the latest version of Go, you can refer to the official documentation at <https://golang.org/doc/install>.

Linux

To install Go in Linux you have two options:

- **Easy option:** Use your distribution package manager:
 - RHEL/Fedora/Centos users with YUM/DNF: `sudo yum install -y golang`
 - Ubuntu/Debian users using APT with: `sudo apt-get install -y golang`
- **Advanced:** Downloading the latest distribution from <https://golang.org>.

I recommend using the second and downloading a distribution. Go's updates maintains backward compatibility and you usually should not be worried about updating your Go binaries frequently.

Go Linux advanced installation

The advanced installation of Go in Linux requires you to download the binaries from **golang** webpage. After entering <https://golang.org>, click the **Download Go** button (usually at the right) some **Featured Downloads** option is available for each distribution. Select **Linux** distribution to download the latest stable version.



At <https://golang.org> you can also download beta versions of the language.

Let's say we have saved the `.tar.gz` file in Downloads folder so let's extract it and move it to a different path. By convention, Go binaries are usually placed in `/usr/local/go` directory:

```
tar -zxvf go*.*.*.linux-amd64.tar.gz  
sudo mv go /usr/local/go
```

On extraction remember to replace asterisks (*) with the version you have downloaded.

Now we have our Go installation in `/usr/local/go` path so now we have to add the `bin` subfolder to our PATH and the `bin` folder within our GOPATH.

```
mkdir -p $HOME/go/bin
```

With `-p` we are telling bash to create all directories that are necessary. Now we need to append bin folder paths to our PATH, append the following lines at the end of your `~/.bashrc`:

```
export PATH=$PATH:/usr/local/go/bin
```

Check that our `go/bin` directory is available:

```
$ go version  
Go version go1.6.2 linux/amd64
```

Windows

To install Go in Windows, you will need administrator privileges. Open your favorite browser and navigate to <https://golang.org>. Once there click the **Download Go** button and select **Microsoft Windows** distribution. A `*.msi` file will start downloading.

Execute the MSI installer by double clicking it. An installer will appear asking you to accept the **End User License Agreement (EULA)** and select a target folder for your installation. We will continue with the default path that in my case was `C:\Go`.

Once the installation is finished you will have to add the **binary Go** folder, located in `C:\Go\bin` to your Path. For this, you must go to Control Panel and select **System** option. Once in System, select the **Advanced** tab and click the **Environment variables** button. Here you'll find a window with variables for your current user and system variables. In system variables, you'll find the **Path** variable. Click it and click the **Edit** button to open a text box. You can add your path by adding `;C:\Go\bin` at the end of the current line (note the semicolon at the beginning of the path). In recent Windows versions (Windows 10) you will have a manager to add variables easily.

Mac OS X

In Mac OS X the installation process is very similar to Linux. Open your favorite browser and navigate to <https://golang.org> and click the **Download Go**. From the list of possible distributions that appear, select **Apple OS X**. This will download a `*.pkg` file to your download folder.

A window will guide you through the installation process where you have to type your administrator password so that it can put Go binary files in /usr/local/go/bin folder with the proper permissions. Now, open **Terminal** to test the installation by typing this on it:

```
$ go version  
Go version go1.6.2 darwin/amd64
```

If you see the installed version, everything was fine. If it doesn't work check that you have followed correctly every step or refer to the documentation at <https://golang.org>.

Setting the workspace - Linux and Apple OS X

Go will always work under the same workspace. This helps the compiler to find packages and libraries that you could be using. This workspace is commonly called **GOPATH**.

GOPATH has a very important role in your working environment while developing Go software. When you import a library in your code it will search for this library in your \$GOPATH/src. The same when you install some Go apps, binaries will be stored in \$GOPATH/bin.

At the same, all your source code must be stored in a valid route within \$GOPATH/src folder. For example, I store my projects in GitHub and my username is *Sayden* so, for a project called **minimal-mesos-go-framework** I will have this folder structure like \$GOPATH/src/github.com/sayden/minimal-mesos-go-framework which reflects the URI where this repo is stored at GitHub:

```
mkdir -p $HOME/go
```

The \$HOME/go path is going to be the destination of our \$GOPATH. We have to set an environment variable with our \$GOPATH pointing to this folder. To set the environment variable, open again the file \$HOME/.bashrc with your favorite text editor and add the following line at the end of it:

```
export GOPATH=${HOME}/go
```

Save the file and open a new terminal. To check that everything is working, just write an echo to the \$GOPATH variable like this:

```
echo $GOPATH/home/mcastro/go
```

If the output of the preceding command points to your chosen Go path, everything is correct and you can continue to write your first program.

Starting with Hello World

This wouldn't be a good book without a Hello World example. Our Hello World example can't be simpler, open your favorite text editor and create a file called `main.go` within our `$GOPATH/src/[your_name]/hello_world` with the following content:

```
package main

func main() {
    println("Hello World!")
}
```

Save the file. To run our program, open the Terminal window of your operating system:

- In Linux, go to programs and find a program called **Terminal**.
- In Windows, hit Windows + R, type `cmd` without quotes on the new window and hit *Enter*.
- In Mac OS X, hit Command + Space to open a spotlight search, type `terminal` without quotes. The terminal app must be highlighted so hit Enter.

Once we are in our terminal, navigate to the folder where we have created our `main.go` file. This should be under your `$GOPATH/src/[your_name]/hello_world` and execute it:

```
go run main.go
Hello World!
```

That's all. The `go run [file]` command will compile and execute our application but it won't generate an executable file. If you want just to build it and get an executable file, you must build the app using the following command:

```
go build -o hello_world
```

Nothing happens. But if you search in the current directory (`ls` command in Linux and Mac OS X; and `dir` in Windows), you'll find an executable file with the name `hello_world`. We have given this name to the executable file when we wrote `-o hello_world` command while building. You can now execute this file:

```
/hello_world
Hello World!
```

And our message appeared! In Windows, you just need to type the name of the `.exe` file to get the same result.



The `go run [my_main_file.go]` command will build and execute the app without intermediate files. The `go build -o [filename]` command will create an executable file that I can take anywhere and has no dependencies.

Integrated Development Environment - IDE

An **IDE (Integrated Development Environment)** is basically a user interface to help developers, code their programs by providing a set of tools to speed up common tasks during development process like compiling, building, or managing dependencies. The IDEs are powerful tools that take some time to master and the purpose of this book is not to explain them (an IDE like Eclipse has its own books).

In Go, you have many options but there are only two that are fully oriented to Go development **LiteIDE** and **IntelliJ Gogland**. LiteIDE is not the most powerful though but IntelliJ has put lots of efforts to make Gogland a very nice editor with completion, debugging, refactoring, testing, visual coverage, inspections, etc. Common IDEs or text editors that have a Go plugin/integration are as following:

- IntelliJ Idea
- Sublime Text 2/3
- Atom
- Eclipse

But you can also find Go plugins for:

- Vim
- Visual Studio and Visual Code

The IntelliJ Idea and Atom IDEs, for the time of this writing, has the support for debugging using a plugin called **Delve**. The IntelliJ Idea is bundled with the official Go plugin. In Atom you'll have to download a plugin called **Go-plus** and a debugger that you can find searching the word `Delve`.

Types

Types give the user the ability to store values in mnemonic names. All programming languages have types related with numbers (to store integers, negative numbers, or floating point for example) with characters (to store a single character) with strings (to store complete words) and so on. Go language has the common types found in most programming languages:

- The `bool` keyword is for Boolean type which represents a `True` or `False` state.
- Many numeric types being the most common:
 - The `int` is a signed integer type, so `int` type represents a number from -2147483648 to 2147483647 in 32 bits machines.
 - The `byte` type represents a number from 0 to 255.
 - The `float32` and `float64` types are the set of all IEEE-754 64/-bit floating-point numbers respectively.
 - You also have `signed int` type like `rune` which is an alias of `int32` type, a number that goes from -2147483648 to 2147483647 and `complex64` and `complex128` which are the set of all complex numbers with `float32/ float64` real and imaginary parts like `2.0i`.
- The `string` keyword for string type represents an array of characters enclosed in quotes like "golang" or "computer".
- An array that is a numbered sequence of elements of a single type and a fixed size (more about arrays later in this chapter). A list of numbers or lists of words with a fixed size is considered arrays.
- The `slice` type is a segment of an underlying array (more about this later in this chapter). This type is a bit confusing at the beginning because it seems like an array but we will see that actually, they are more powerful.
- The structures that are the objects that are composed of another objects or types.
- The pointers (more about this later in this chapter) are like directions in the memory of our program (yes, like mailboxes that you don't know what's inside).
- The functions are interesting (more about this later in this chapter). You can also define functions as variables and pass them to other functions (yes, a function that uses a function, did you like Inception movie?).
- The `interface` is incredibly important for the language as they provide many encapsulation and abstraction functionalities that we'll need often. We'll use interfaces extensively during the book and they are presented in greater detail later.

- The map types are unordered key-value structures. So for a given key, you have an associated value.
- The channels are the communication primitive in Go for concurrency programs. We'll look on channels with more detail on Chapter 8, *Dealing with Go's CSP concurrency*.

Variables and constants

Variables are spaces in computer's memory to store values that can be modified during the execution of the program. Variables and constants have a type like the ones described in preceding text. Although, you don't need to explicitly write the type of them (although you can do it). This property to avoid explicit type declaration is what is called **Inferred types**. For example:

```
//Explicitly declaring a "string" variable
var explicit string = "Hello, I'm a explicitly declared variable"
```

Here we are declaring a variable (with the keyword var) called `explicit` of string type. At the same time, we are defining the value to Hello World!.

```
//Implicitly declaring a "string". Type inferred
inferred := ", I'm an inferred variable "
```

But here we are doing exactly the same thing. We have avoided the `var` keyword and the `string` type declaration. Internally, Go's compiler will infer (guess) the type of the variable to a string type. This way you have to write much less code for each variable definition.

The following lines use the `reflect` package to gather information about a variable. We are using it to print the type of (the `TypeOf` variable in the code) of both variables:

```
fmt.Println("Variable 'explicit' is of type:",
            reflect.TypeOf(explicit))
fmt.Println("Variable 'inferred' is of type:",
            reflect.TypeOf(inferred))
```

When we run the program, the result is the following:

```
$ go run main.go
Hello, I'm a explicitly declared variable
Hello, I'm an inferred variable
Variable 'explicit' is of type: string
Variable 'inferred' is of type: string
```

As we expected, the compiler has inferred the type of the implicit variable to string too. Both have written the expected output to the console.

Operators

The operators are used to perform arithmetic operations and make comparisons between many things. The following operators are reserved by Go language.

+	&	+=	&=	&&	==	!=	()
-		-=	=		<	<=	[]
*	^	*=	^=	<-	>	>=	{	}
/	<<	/=	<<=	++	=	:=	,	;
%	>>	%=	>>=	--	!	:
	&&		&&=					

Most commonly used operators are the arithmetic operators and comparators. Arithmetic operators are as following:

- The + operator for sums
- The - operator for subtractions
- The * operator for multiplications
- The / operator for divisions
- The % operator for division remainders
- The ++ operator to add 1 to the current variable
- The -- operator to subtract 1 to the current variable

On the other side, comparators are used to check the differences between two statements:

- The == operator to check if two values are equal
- The != operator to check if two values are different
- The > operator to check if left value is higher than right value
- The < operator to check if left value is lower than right value
- The >= operator to check if left value is higher or equal to right value
- The <= operator to check if left value is lower or equal to right value
- The && operator to check if two values are true

You also have the shifters to perform a binary shift to left or right of a value and a negated operator to invert some value. We'll use these operators a lot during the following chapters so don't worry too much about them now, just keep in mind that you cannot set the name of any variable, field or function in your code like this operators.



What's the inverted value of 10? What's the negated value of 10? -10?
Incorrect.. 10 in binary code is 1010 so if we negate every number we will have 0101 or 101 which is the number 5.

Flow control

Flow control is referred as the ability to decide which portion of code or how many times you execute some code on a condition. In Go, it is implemented using familiar imperative clauses like if, else, switch and for. The syntax is easy to grasp. Let's review major flow control statements in Go.

The if... else statement

Go language, like most programming languages, has if...else conditional statement for flow control. The Syntax is similar to other languages but you don't need to encapsulate the condition between parenthesis:

```
ten := 10
if ten == 20 {
    println("This shouldn't be printed as 10 isn't equal to 20")
} else {
    println("Ten is not equals to 20");
}
```

The else...if condition works in a similar fashion, you don't need parentheses either and they are declared as programmer would expect:

```
if "a" == "b" || 10 == 10 || true == false {
    println("10 is equal to 10")
} else if 11 == 11 && "go" == "go" {
    println("This isn't print because previous condition was satisfied");
} else {
    println("In case no condition is satisfied, print this")
}
```



Go does not have ternary conditions like `condition ? true : false`.

The switch statement

The `switch` statement is also similar to most imperative languages. You take a variable and check possible values for it:

```
number := 3
switch(number){
    case 1:
        println("Number is 1")
    case 2:
        println("Number is 2")
    case 3:
        println("Number is 3")
}
```

The for...range statement

The `_for_` loop is also similar than in common programming languages but you don't use parentheses either

```
for i := 0; i<=10; i++ {
    println(i)
}
```

As you have probably imagined if you have computer science background, we infer an `int` variable defined as `0` and execute the code between the brackets while the condition (`i<=10`) is satisfied. Finally, for each execution, we added `1` to the value of `i`. This code will print the numbers from `0` to `10`. You also have a special syntax to iterate over arrays or slices which is `range`:

```
for index, value := range my_array {
    fmt.Printf("Index is %d and value is %d", index, value)
}
```

First, the `fmt` (format) is a very common Go package that we will use extensively to give shape to the message that we will print in the console.

Regarding for, you can use the range keyword to retrieve every item in a collection like my_array and assign them to the value temporal variable. It will also give you an index variable to know the position of the value you're retrieving. It's equivalent to write the following:

```
for index := 0, index < len(my_array); index++ {  
    value := my_array[index]  
    fmt.Printf("Index is %d and value is %d", index, value)  
}
```



The len method is used to know the length of a collection.

If you execute this code, you'll see that the result is the same.

Functions

A function is a small portion of code that surrounds some action you want to perform and returns one or more values (or nothing). They are the main tool for developer to maintain structure, encapsulation, and code readability but also allow an experienced programmer to develop proper unit tests against his or her functions.

Functions can be very simple or incredibly complex. Usually, you'll find that simpler functions are also easier to maintain, test and debug. There is also a very good advice in computer science world that says: *A function must do just one thing, but it must do it damn well.*

What does a function look like?

A function is a piece of code with its own variables and flow that doesn't affect anything outside of the opening and close brackets but global package or program variables.

Functions in Go has the following composition:

```
func [function_name] (param1 type, param2 type...) (returned type1,  
returned type2...) {  
    //Function body  
}
```

Following the previous definition, we could have the following example:

```
func hello(message string) error {
```

```
    fmt.Sprintf("Hello %s\n", message)
    return nil
}
```

Functions can call other functions. For example, in our previous `hello` function, we are receiving a message argument of type string and we are calling a different function `fmt.Sprintf("Hello %s\n", message)` with our argument as parameter. Functions can also be used as parameters when calling other functions or be returned.

It is very important to choose a good name for your function so that it is very clear what it is about without writing too many comments over it. This can look a bit trivial but choosing a good name is not so easy. A short name must show what the function does and let the reader imagine what error is it handling or if it's doing any kind of logging. Within your function, you want to do everything that a particular behavior need but also to control expected errors and wrapping them properly.

So, to write a function is more than simply throw a couple of lines that does what you need, that's why it is important to write a unit test, make them small and concise.

What is an anonymous function?

An anonymous function is a function without a name. This is useful when you want to return a function from another function that doesn't need a context or when you want to pass a function to a different function. For example, we will create a function that accepts one number and returns a function that accepts a second number that it adds it to the first one. The second function does not have a declarative name (as we have assigned it to a variable) that is why it is said to be anonymous:

```
func main() {
    add := func(m int) {
        return m+1
    }

    result := add(6)

    //1 + 6 must print 7
    println(result)
}
```

The `add` variable points to an anonymous function that adds one to the specified parameter. As you can see, it can be used only for the scope its parent function `main` and cannot be called from anywhere else.

Anonymous functions are really powerful tools that we will use extensively on design patterns.

Closures

Closures are something very similar to anonymous functions but even more powerful. The key difference between them is that an anonymous function has no context within itself and a closure has. Let's rewrite the previous example to add an arbitrary number instead of one:

```
func main() {
    addN := func(m int) {
        return func(n int) {
            return m+n
        }
    }

    addFive := addN(5)
    result := addN(6)
    //5 + 6 must print 7

    println(result)
}
```

The `addN` variable points to a function that returns another function. But the returned function has the context of the `m` parameter within it. Every call to `addN` will create a new function with a fixed `m` value, so we can have main `addN` functions, each adding a different value.

This ability of closures is very useful to create libraries or deal with functions with unsupported types.

Creating errors, handling errors and returning errors.

Errors are extensively used in Go, probably thanks to its simplicity. To create an error simply make a call to `errors.New(string)` with the text you want to create on the error. For example:

```
err := errors.New("Error example")
```

As we have seen before, we can return errors to a function. To handle an error you'll see the following pattern extensively in Go code:

```
func main() {
    err := doesReturnError()
    if err != nil {
        panic(err)
    }
}

func doesReturnError() error {
    err := errors.New("this function simply returns an error")
    return err
}
```

Function with undetermined number of parameters

Functions can be declared as *variadic*. This means that its number of arguments can vary. What this does is to provide an array to the scope of the function that contains the arguments that the function was called with. This is convenient if you don't want to force the user to provide an array when using this function. For example:

```
func main() {
    fmt.Printf("%d\n", sum(1,2,3))
    fmt.Printf("%d\n", sum(4,5,6,7,8))
}

func sum(args ...int) (result int) {
    for _, v := range args {
        result += v
    }
    return
}
```

In this example, we have a `sum` function that will return the sum of all its arguments but take a closer look at the `main` function where we call `sum`. As you can see now, first we call `sum` with three arguments and then with five arguments. For `sum` functions, it doesn't matter how many arguments you pass as it treats its arguments as an array all in all. So on our `sum` definition, we simply iterate over the array to add each number to the `result` integer.

Naming returned types

Have you realized that we have given a name to the returned type? Usually, our declaration would be written as `func sum(args int) int` but you can also name the variable that you'll use within the function as a return value. Naming the variable in the return type would also zero-value it (in this case, an `int` will be initialized as zero). At the end, you just need to return the function (without value) and it will take the respective variable from the scope as returned value. This also makes easier to follow the mutation that the returning variable is suffering as well as to ensure that you aren't returning a mutated argument.

Arrays, slices, and maps

Arrays are one of the most widely used types of computer programming. They are lists of other types that you can access by using their position on the list. The only downside of an array is that its size cannot be modified. Slices allow the use of arrays with variable size. The `maps` type will let us have a dictionary like structures in Go. Let's see how each work.

Arrays

An array is a numbered sequence of elements of a single type. You can store 100 different unsigned integers in a unique variable, three strings or 400 `bool` values. Their size cannot be changed.

You must declare the length of the array on its creation as well as the type. You can also assign some value on creation. For example here you have 100 `int` values all with 0 as value:

```
var arr [100]int
```

Or an array of size 3 with `strings` already assigned:

```
arr := [3]string{"go", "is", "awesome"}
```

Here you have an array of 2 `bool` values that we initialize later:

```
var arr [2]bool
arr[0] = true
arr[1] = false
```

Zero-initialization

In our previous example, we have initialized an `array` of `bool` values of size 2. We wouldn't need to assign `arr[1]` to `false` because of the nature of zero-initialization in the language. Go will initialize every value in a `bool` array to `false`. We will look deeper to zero-initialization later in this chapter.

Slices

Slices are similar to arrays, but their size can be altered on runtime. This is achieved, thanks to the underlying structure of a slice that is an array. So, like arrays, you have to specify the type of the slice and its size. So, use the following line to create a slice:

```
mySlice := make([]int, 10)
```

This command has created an underlying array of ten elements. If we need to change the size of the slice by, for example, adding a new number, we would append the number to the slice:

```
mySlice := append(mySlice, 5)
```

The syntax of `append` is of the form (`[array to append an item to], [item to append]`) and returns the new slice, it does not modify the actual slice. This is also true to delete an item. For example, let's delete the first item of the array as following:

```
mySlice := mySlice[1:]
```

Yes, like in arrays. But what about deleting the second item? We use the same syntax:

```
mySlice = append(mySlice[:1], mySlice[2:]...)
```

We take all elements from zero index (included) to the first index (not included) and each element from the second index (included) to the end of the array, effectively deleting the value at the second position in the slice (index 1 as we start counting with 0). As you can see, we use the undetermined arguments syntax as the second parameter.

Maps

Maps are like dictionaries--for each word, we have a definition but we can use any type as word or definition and they'll never be ordered alphabetically. We can create maps of string that point to numbers, a string that points to interfaces and structs that point to int and int to function. You cannot use as key: slices, the functions, and maps. Finally, you create maps by using the keyword make and specifying the key type and the value type:

```
myMap := make(map[string]int)
myMap["one"] = 1
myMap["two"] = 2
fmt.Println(myMap["one"])
```

When parsing JSON content, you can also use them to get a string[interface] map:

```
myJsonMap := make(map[string]interface{})
jsonData := []byte(`{"hello":"world"}`)
err := json.Unmarshal(jsonData, &myJsonMap)
if err != nil {
    panic(err)
}
fmt.Sprintf("%s\n", myJsonMap["hello"])
```

The `myJsonMap` variable is a map that will store the contents of JSON and that we will need to pass its pointer to the `Unmarshal` function. The `jsonData` variable declares an array of bytes with the typical content of a JSON object; we are using this as the mock object. Then, we unmarshal the contents of the JSON storing the result of the memory location of `myJsonMap` variable. After checking that the conversion was ok and the JSON byte array didn't have syntax mistakes, we can access the contents of the map in a JSON-like syntax.

Visibility

Visibility is the attribute of a function or a variable to be visible to different parts of the program. So a variable can be used only in the function that is declared, in the entire package or in the entire program.

How can I set the visibility of a variable or function? Well, it can be confusing at the beginning but it cannot be simpler:

- Uppercase definitions are public (visible in the entire program).
- Lowercase are private (not seen at the package level) and function definitions (variables within functions) are visible just in the scope of the function.

Here you can see an example of a public function:

```
package hello

func Hello_world() {
    println("Hello World!")
}
```

Here, `Hello_world` is a global function (a function that is visible across the entire source code and to third party users of your code). So, if our package is called `hello`, we could call this function from outside of this package by using `hello.Hello_world()` method.

```
package different_package

import "github.com/sayden/go-design-patterns/first_chapter/hello"

func myLibraryFunc() {
    hello.Hello_world()
}
```

As you can see, we are in the `different_package` package. We have to import the package we want to use with the keyword `import`. The route then is the path within your `$GOPATH/src` that contains the package we are looking for. This path conveniently matches the URL of a GitHub account or any other **Concurrent Versions System(CVS)** repository.

Zero-initialization

Zero-initialization is a source of confusion sometimes. They are default values for many types that are assigned even if you don't provide a value for the definition. Following are the zero-initialization for various types:

- The `false` initialization for `bool` type.
- Using `0` values for `int` type.
- Using `0.0` for `float` type.
- Using `" "` (empty strings) for `string` type.
- Using `nil` keyword for pointers, functions, interfaces, slices, channels and maps.
- Empty `struct` for structures without fields.
- Zero-initialized `struct` for structures with fields. The zero value of a structure is defined as the structure that has its fields initialized as zero value too.

Zero-initialization is important when programming in Go because you won't be able to return a `nil` value if you have to return an `int` type or a `struct`. Keep this in mind, for example, in functions where you have to return a `bool` value. Imagine that you want to know if a number is divisible by a different number but you pass `0` (zero) as the divisor.

```
func main() {
    res := divisibleBy(10, 0)
    fmt.Printf("%v\n", res)
}

func divisibleBy(n, divisor int) bool {
    if divisor == 0 {
        //You cannot divide by zero
        return false
    }

    return (n % divisor == 0)
}
```

The output of this program is `false` but this is incorrect. A number divided by zero is an error, it's not that 10 isn't divisible by zero but that a number cannot be divided by zero by definition. Zero-initialization is making things awkward in this situation. So, how can we solve this error? Consider the following code:

```
func main() {
    res, err := divisibleBy(10, 0)
    if err != nil {
        log.Fatal(err)
    }

    log.Printf("%v\n", res)
}

func divisibleBy(n, divisor int) (bool, error) {
    if divisor == 0 {
        //You cannot divide by zero
        return false, errors.New("A number cannot be divided by zero")
    }

    return (n % divisor == 0), nil
}
```

We're dividing 10 by 0 again but now the output of this function is `A number cannot be divided by zero`. Error captured, the program finished gracefully.

Pointers and structures

Pointers are the number one source of a headache of every C or C++ programmer. But they are one of the main tools to achieve high-performance code in non-garbage-collected languages. Fortunately for us, Go's pointers have achieved the best of both worlds by providing high-performance pointers with garbage-collector capabilities and easiness.

On the other side for its detractors, Go lacks inheritance in favor of composition. Instead of talking about the objects that *are* in Go, your objects *have* other. So, instead of having a `car` structure that inherits the class `vehicle` (a car is a vehicle), you could have a `vehicle` structure that contains a `car` structure within.

What is a pointer? Why are they good?

Pointers are hated, loved, and very useful at the same time. To understand what a pointer is can be difficult so let's try with a real world explanation. As we mentioned earlier in this chapter, a pointer is like a mailbox. Imagine a bunch of mailboxes in a building; all of them have the same size and shape but each refers to a different house within the building. Just because all mailboxes are the same size does not mean that each house will have the same size. We could even have a couple of houses joined, a house that was there but now has a license of commerce, or a house that is completely empty. So the pointers are the mailboxes, all of them of the same size and that refer to a house. The building is our memory and the houses are the types our pointers refer to and the memory they allocate. If you want to receive something in your house, it's far easier to simply send the address of your house (to send the pointer) instead of sending the entire house so that your package is deposited inside. But they have some drawbacks as if you send your address and your house (variable it refers to) disappears after sending, or its type owner changes--you'll be in trouble.

How is this useful? Imagine that somehow you have 4 GB of data in a variable and you need to pass it to a different function. Without a pointer, the entire variable is cloned to the scope of the function that is going to use it. So, you'll have 8 GB of memory occupied by using this variable twice that, hopefully, the second function isn't going to use in a different function again to raise this number even more.

You could use a pointer to pass a very small reference to this chunk to the first function so that just the small reference is cloned and you can keep your memory usage low.

While this isn't the most academic nor exact explanation, it gives a good view of what a pointer is without explaining what a stack or a heap is or how they work in x86 architectures.

Pointers in Go are very limited compared with C or C++ pointers. You can't use pointer arithmetic nor can you create a pointer to reference an exact position in the stack.

Pointers in Go can be declared like this:

```
number := 5
```

Here `number := 5` code represents our 4 GB variable and `pointer_to_number` contains the reference (represented by an ampersand) to this variable. It's the direction to the variable (the one that you put in the mailbox of this house/type/variable). Let's print the variable `pointer_to_number`, which is a simple variable:

```
println(pointer_to_number)
0x005651FA
```

What's that number? Well, the direction to our variable in memory. And how can I print the actual value of the house? Well, with an asterisk (*) we tell the compiler to take the value that the pointer is referencing, which is our 4 GB variable.

```
println(*pointer_to_number)
5
```

Structs

A struct is an object in Go. It has some similarities with classes in OOP as they have fields. Structs can implement interfaces and declare methods. But, for example, in Go, there's not inheritance. Lack of inheritance looks limiting but in fact, *composition over inheritance* was a requirement of the language.

To declare a structure, you have to prefix its name with the keyword `type` and suffix with the keyword `struct` and then you declare any field or method between brackets, for example:

```
type Person struct {
    Name string
    Surname string
    Hobbies []string
    id string
}
```

In this piece of code, we have declared a `Person` structure with three public fields (`Name`, `Age`, and `Hobbies`) and one private field (`id`, if you recall the *Visibility* section in this chapter, lowercase fields in Go refers to private fields are just visible within the same package). With this `struct`, we can now create as many instances of `Person` as we want. Now we will write a function called `GetFullName` that will give the composition of the name and the surname of the `struct` it belongs to:

```
func (person *Person) GetFullName() string {
    return fmt.Sprintf("%s %s", person.Name, person.Surname)
}

func main() {
    p := Person{
        Name: "Mario",
        Surname: "Castro",
        Hobbies: []string{"cycling", "electronics", "planes"},
        id: "sa3-223-asd",
    }

    fmt.Printf("%s likes %s, %s and %s\n", p.GetFullName(), p.Hobbies[0],
    p.Hobbies[1], p.Hobbies[2])
}
```

Methods are defined similarly to functions but in a slightly different way. There is a (`p *Person`) that refers to a pointer to the created instance of the `struct` (recall the *Pointers* section in this chapter). It's like using the keyword `this` in Java or `self` in Python when referring to the pointing object.

Maybe you are thinking why does (`p *Person`) have the pointer operator to reflect that `p` is actually a pointer and not a value? This is because you can also pass `Person` by value by removing the pointer signature, in which case a copy of the value of `Person` is passed to the function. This has some implications, for example, any change that you make in `p` if you pass it by value won't be reflected in source `p`. But what about our `GetFullName()` method?

```
func (person Person) GetFullName() string {
    return fmt.Sprintf("%s %s", person.Name, person.Surname)
}
```

Its console output has no effect in appearance but a full copy was passed before evaluating the function. But if we modify `person` here, the source `p` won't be affected and the new `person` value will be available only on the scope of this function.

On the `main` function, we create an instance of our structure called `p`. As you can see, we have used implicit notation to create the variable (the `:=` symbol). To set the fields, you have to refer to the name of the field, colon, the value, and the comma (don't forget the comma at the end!). To access the fields of the instantiated structure, we just refer to them by their name like `p.Name` or `p.Surname`. You use the same syntax to access the methods of the structure like `p.GetFullName()`.

The output of this program is:

```
$ go run main.go
Mario Castro likes cycling, electronics and planes
```

Structures can also contain another structure (composition) and implement interface methods apart from their own but, what's an interface method?

Interfaces

Interfaces are essential in object-oriented programming, in functional programming (traits) and, especially, in design patterns. Go's source code is full of interfaces everywhere because they provide the abstraction needed to deliver uncoupled code with the help of functions. As a programmer, you also need this type of abstraction when you write libraries but also when you write code that is going to be maintained in the future with new functionality.

Interfaces are something difficult to grasp at the beginning but very easy once you have understood their behavior and provide very elegant solutions for common problems. We will use them extensively during this book so put special focus on this section.

Interfaces - signing a contract

An interface is something really simple but powerful. It's usually defined as a contract between the objects that implement it but this explanation isn't clear enough in my honest opinion for newcomers to the interface world.

A water-pipe is a contract too; whatever you pass through it must be a liquid. Anyone can use the pipe, and the pipe will transport whatever liquid you put in it (without knowing the content). The water-pipe is the interface that enforces that the users must pass liquids (and not something else).

Let's think about another example: a train. The railroads of a train are like an interface. A train must construct (implement) its width with a specified value so that it can enter the railroad but the railroad never knows exactly what it's carrying (passengers or cargo). So for example, an interface of the railroad will have the following aspect:

```
type RailroadWideChecker interface {
    CheckRailsWidth() int
}
```

The `RailroadWideChecker` is the type our trains must implement to provide information about their width. The trains will verify that the train isn't too wide or too narrow to use its railroads:

```
type Railroad struct {
    Width int
}

func (r *Railroad) IsCorrectSizeTrain(r RailroadWideChecker) bool {
    return r.CheckRailsWidth() != r.Width
}
```

The `Railroad` is implemented by an imaginary station object that contains the information about the width of the railroads in this station and that has a method to check whether a train fits the needs of the railroad with the `IsCorrectSizeTrain` method. The `IsCorrectSizeTrain` method receives an interface object which is a pointer to a train that implements this interface and returns a validation between the width of the train and the railroad:

```
Type Train struct {
    TrainWidth int
}

func (p *Train) CheckRailsWidth() int {
    return p.TrainWidth
}
```

Now we have created a passenger's train. It has a field to contain its width and implements our `CheckRailsWidth` interface method. This structure is considered to fulfill the needs of a `RailRoadWideChecker` interface (because it has an implementation of the methods that the interfaces ask for).

So now, we'll create a railroad of 10 units wide and two trains--one of 10 units wide that fit the railroad size and another of 15 units that cannot use the railroad.

```
func main() {
    railroad := Railroad{Width:10}
```

```
passengerTrain := Train{TrainWidth: 10}
cargoTrain := Train {TrainWidth: 15}

canPassengerTrainPass := railroad.IsCorrectSizeTrain(passengerTrain)
canCargoTrainPass := railroad.IsCorrectSizeTrain(cargoTrain)

fmt.Printf("Can passenger train pass? %b\n", canPassengerTrainPass)
fmt.Printf("Can cargo train pass? %b\n", canCargoTrainPass)
}
```

Let's dissect this `main` function. First, we created a `railroad` object of 10 units called `railroad`. Then two trains, of 10 and 15 units' width for passengers and cargo respectively. Then, we pass both objects to the `railroad` method that accepts interfaces of the `RailroadWideChecker` interface. The `railroad` itself does not know the width of each train separately (we'll have a huge list of trains) but it has an interface that trains must implement so that it can ask for each width and returns a value telling you if a train can or cannot use of the railroads. Finally, the output of the call to `printf` function is the following:

```
Can passenger train pass? true
Can cargo train pass? false
```

As I mentioned earlier, interfaces are so widely used during this book that it doesn't matter if it still looks confusing for the reader as they'll be plenty of examples during the book.

Testing and TDD

When you write the first lines of some library, it's difficult to introduce many bugs. But once the source code gets bigger and bigger, it becomes easier to break things. The team grows and now many people are writing the same source code, new functionality is added on top of the code that you wrote at the beginning. And code stopped working by some modification in some function that now nobody can track down.

This is a common scenario in enterprises that testing tries to reduce (it doesn't completely solve it, it's not a holy grail). When you write unit tests during your development process, you can check whether some new feature is breaking something older or whether your current new feature is achieving everything expected in the requirements.

Go has a powerful testing package that allows you also to work in a TDD environment quite easily. It is also very convenient to check the portions of your code without the need to write an entire main application that uses it.

The testing package

Testing is very important in every programming language. Go creators knew it and decided to provide all libraries and packages needed for the test in the core package. You don't need any third-party library for testing or code coverage.

The package that allows for testing Go apps is called, conveniently, testing. We will create a small app that sums two numbers that we provide through the command line:

```
func main() {
    //Atoi converts a string to an int
    a, _ := strconv.Atoi(os.Args[1])
    b, _ := strconv.Atoi(os.Args[2])

    result := sum(a,b)
    fmt.Printf("The sum of %d and %d is %d\n", a, b, result)
}

func sum(a, b int) int {
    return a + b
}
```

Let's execute our program in the terminal to get the sum:

```
$ go run main.go 3 4
The sum of 3 and 4 is 7
```

By the way, we're using the `strconv` package to convert strings to other types, in this case, to `int`. The method `Atoi` receives a string and returns an `int` and an `error` that, for simplicity, we are ignoring here (by using the underscore).



You can ignore variable returns by using the underscores if necessary, but usually, you don't want to ignore errors.

Ok, so let's write a test that checks the correct result of the sum. We're creating a new file called `main_test.go`. By convention, test files are named like the files they're testing plus the `_test` suffix:

```
func TestSum(t *testing.T) {
    a := 5
    b := 6
    expected := 11

    res := sum(a, b)
```

```
    if res != expected {
        t.Errorf("Our sum function doesn't work, %d+%d isn't %d\n", a, b,
res)
    }
}
```

Testing in Go is used by writing methods started with the prefix `Test`, a test name, and the injection of the `testing.T` pointer called `t`. Contrary to other languages, there are no asserts nor special syntax for testing in Go. You can use Go syntax to check for errors and you call `t` with information about the error in case it fails. If the code reaches the end of the `Test` function without arising errors, the function has passed the tests.

To run a test in Go, you must use the `go test -v` command (`-v` is to receive verbose output from the test) keyword, as following:

```
$ go test -v
==== RUN TestSum
--- PASS: TestSum (0.00s)
PASS
ok  github.com/go-design-patterns/introduction/ex_xx_testing 0.001s
```

Our tests were correct. Let's see what happens if we break things on purpose and we change the expected value of the test from 11 to 10:

```
$ go test
--- FAIL: TestSum (0.00s)
    main_test.go:12: Our sum function doesn't work, 5+6 isn't 10
FAIL
exit status 1
FAIL  github.com/sayden/go-design-patterns/introduction/ex_xx_testing
0.002s
```

The test has failed (as we expected). The testing package provides the information you set on the test. Let's make it work again and check test coverage. Change the value of the variable `expected` from 10 to 11 again and run the command `go test -cover` to see code coverage:

```
$ go test -cover
PASS
coverage: 20.0% of statements
ok  github.com/sayden/go-design-patterns/introduction/ex_xx_testing 0.001s
```

The `-cover` options give us information about the code coverage for a given package. Unfortunately, it doesn't provide information about overall application coverage.

What is TDD?

TDD is the acronym for **Test Driven Development**. It consists of writing the tests first before writing the function (instead of what we did just before when we wrote the `sum` function first and then we wrote the `test` function).

TDD changes the way to write code and structure code so that it can be tested (a lot of code you can find in GitHub, even code that you have probably written in the past is probably very difficult, if not impossible, to test).

So, how does it work? Let's explain this with a real life example--imagine that you are in summer and you want to be refreshed somehow. You can build a pool, fill it with cold water, and jump into it. But in TDD terms, the steps will be:

1. You jump into a place where the pool will be built (you write a test that you know it will fail).
2. It hurts... and you aren't cool either (yes... the test failed, as we predicted).
3. You build a pool and fill it with cold water (you code the functionality).
4. You jump into the pool (you repeat the point 1 test again).
5. You're cold now. Awesome! Object completed (test passed).
6. Go to the fridge and take a beer to the pool. Drink. Double awesomeness (refactor the code).

So let's repeat the previous example but with a multiplication. First, we will write the declaration of the function that we're going to test:

```
func multiply(a, b int) int {  
    return 0  
}
```

Now let's write the test that will check the correctness of the previous function:

```
import "testing"  
  
func TestMultiply(t *testing.T) {  
    a := 5  
    b := 6  
    expected := 30  
  
    res := multiply(a, b)  
    if res != expected {  
        t.Errorf("Our multiply function doesn't work, %d * %d isn't %d\n", a,  
b, res)  
    }  
}
```

}

And we test it through the command line:

```
$ go test
--- FAIL: TestMultiply (0.00s)
main_test.go:12: Our multiply function doesn't work, 5+6 isn't 0
FAIL
exit status 1
FAIL    github.com/sayden/go-
designpatterns/introduction/ex_xx_testing/multiply
0.002s
```

Nice. Like in our pool example where the water wasn't there yet, our function returns an incorrect value too. So now we have a function declaration (but isn't defined yet) and the test that fails. Now we have to make the test pass by writing the function and executing the test to check:

```
func multiply(a, b int) int {
    return a*b
}
```

And we execute again our testing suite. After writing our code correctly, the test should pass so we can continue to the refactoring process:

```
$ go test
PASS
ok    github.com/sayden/go-design-
patterns/introduction/ex_xx_testing/multiply
0.001s
```

Great! We have developed the `multiply` function following TDD. Now we must refactor our code but we cannot make it more simple or readable so the loop can be considered closed.

During this book, we will write many tests that define the functionality that we want to achieve in our patterns. TDD promotes encapsulation and abstraction (just like design patterns do).

Libraries

Until now, most of our examples were applications. An application is defined by its `main` function and package. But with Go, you can also create pure libraries. In libraries, the package need not be called `main` nor do you need the `main` function.

As libraries aren't applications, you cannot build a binary file with them and you need the main package that is going to use them.

For example, let's create an arithmetic library to perform common operations on integers: sums, subtractions, multiplications, and divisions. We'll not get into many details about the implementation to focus on the particularities of Go's libraries:

```
package arithmetic

func Sum(args ...int) (res int) {
    for _, v := range args {
        res += v
    }
    return
}
```

First, we need a name for our library; we set this name by giving a name to the entire package. This means that every file in this folder must have this package name too and the entire group of files composes the library called **arithmetic** too in this case (because it only contains one package). This way, we won't need to refer to the filenames for this library and to provide the library name and path will be enough to import and use it. We have defined a `Sum` function that takes as many arguments as you need and that will return an integer that, during the scope of the function, is going to be called `res`. This allows us to initialize to 0 the value we're returning. We defined a package (not the `main` package but a library one) and called it `arithmetic`. As this is a library package, we can't run it from the command line directly so we'll have to create the `main` function for it or a unit test file. For simplicity, we'll create a `main` function that runs some of the operations now but let's finish the library first:

```
func Subtract(args ...int) int {
    if len(args) < 2 {
        return 0
    }

    res := args[0]
    for i := 1; i < len(args); i++ {
        res -= args[i]
    }
    return res
}
```

The `Subtraction` code will return 0 if the number of arguments is less than zero and the subtraction of all its arguments if it has two arguments or more:

```
func Multiply(args ...int) int {
```

```
if len(args) < 2 {
    return 0
}

res := 1
for i := 0; i < len(args); i++ {
    res *= args[i]
}
return res
}
```

The `Multiply` function works in a similar fashion. It returns 0 when arguments are less than two and the multiplication of all its arguments when it has two or more. Finally, the `Division` code changes a bit because it will return an error if you ask it to divided by zero:

```
func Divide(a, b int) (float64, error) {
    if b == 0 {
        return 0, errors.New("You cannot divide by zero")
    }
    return float64(a) / float64(b), nil
}
```

So now we have our library finished, but we need a `main` function to use it as libraries cannot be converted to executable files directly. Our main function looks like the following:

```
package main

import (
    "fmt"
    "bitbucket.org/mariocastro/go-design-
patterns/introduction/libraries/arithmetic"
)

func main() {
    sumRes := arithmetic.Sum(5, 6)
    subRes := arithmetic.Subtract(10, 5)
    multiplyRes := arithmetic.Multiply(8, 7)
    divideRes, _ := arithmetic.Divide(10, 2)

    fmt.Printf("5+6 is %d. 10-5 is %d, 8*7 is %d and 10/2 is %f\n", sumRes,
    subRes, multiplyRes, divideRes)
}
```

We are performing an operation over every function that we have defined. Take a closer look at the `import` clause. It is taking the library we have written from its folder within `$GOPATH` that matches its URL in <https://bitbucket.org/>. Then, to use every one of the functions that are defined within a library, you have to name the package name that the library has before each method.



Have you realized that we called our functions with uppercase names? Because of the visibility rules we have seen before, exported functions in a package must have uppercase names or they won't be visible outside of the scope of the package. So, with this rule in mind, you cannot call a lowercase function or variable within a package and package calls will always be followed by uppercase names.

Let's recall, some naming conventions about libraries:

- Each file in the same folder must contain the same package name. Files don't need to be named in any special way.
- A folder represents a package name within a library. The folder name will be used on import paths and it doesn't need to reflect the package name (although it's recommended for the parent package).
- A library is one or many packages representing a tree that you import by the parent of all packages folder.
- You call things within a library by their package name.

The Go get tool

Go get is a tool to get third party projects from CVS repositories. Instead of using the `git clone` command, you can use Go get to receive a series of added benefits. Let's write an example using CoreOS's ETCD project which is a famous distributed key-value store.

CoreOS's ETCD is hosted on GitHub at <https://github.com/coreos/etcd.git>. To download this project source code using the Go get tool, we must type in the Terminal its resulting import path that it will have in our GOPATH:

```
$ go get github.com/coreos/etcd
```

Note that we have just typed the most relevant information so that Go get figures out the rest. You'll get some output, depending on the state of the project, but after, while, it will disappear. But what did happen?

- Go get has created a folder in \$GOPATH/src/github.com/coreos.
- It has cloned the project in that location, so now the source code of ETCD is available at \$GOPATH/src/github.com/coreos/etcd.
- Go get has cloned any repository that ETCD could need.
- It has tried to install the project if it is not a library. This means, it has generated a binary file of ETCD and has put it in \$GOPATH/bin folder.

By simply typing the `go get [project]` command, you'll get all that material from a project in your system. Then in your Go apps, you can just use any library by importing the path within the source. So for the ETCD project, it will be:

```
import "github.com/coreos/etcd"
```

It's very important that you get familiar with the use of the Go get tool and stop using `git clone` when you want a project from a Git repository. This will save you some headaches when trying to import a project that isn't contained within your GOPATH.

Managing JSON data

JSON is the acronym for **JavaScript Object Notation** and, like the name implies, it's natively JavaScript. It has become very popular and it's the most used format for communication today. Go has very good support for JSON serialization/deserialization with the `JSON` package that does most of the dirty work for you. First of all, there are two concepts to learn when working with JSON:

- **Marshal:** When you marshal an instance of a structure or object, you are converting it to its JSON counterpart.
- **Unmarshal:** When you are unmarshaling some data, in the form of an array of bytes, you are trying to convert some JSON-expected-data to a known struct or object. You can also *unmarshal* to a `map[string]interface{}` in a fast but not very safe way to interpret the data as we'll see now.

Let's see an example of marshaling a string:

```
import (
    "encoding/json"
    "fmt"
)

func main() {
    packt := "packt"
```

```
    jsonPackt, ok := json.Marshal(packt)
    if !ok {
        panic("Could not marshal object")
    }
    fmt.Println(string(jsonPackt))
}
$ "pack"
```

First, we have defined a variable called `packt` to hold the contents of the `packt` string. Then, we have used the `json` library to use the `Marshal` command with our new variable. This will return a new `bytearray` with the JSON and a flag to provide and `boolOK` result for the operation. When we print the contents of the bytes array (previous casting to `string`) the expected value appears. Note that `packt` appeared actually between quotes as the JSON representation would be.

The encoding package

Have you realized that we have imported the package `encoding/json`? Why is it prefixed with the word `encoding`? If you take a look at Go's source code to the `src/encoding` folder you'll find many interesting packages for encoding/decoding such as, XML, HEX, binary, or even CSV.

Now something a bit more complicated:

```
type MyObject struct {
    Number int
    `json:"number"`
    Word string
}

func main() {
    object := MyObject{5, "Packt"}
    oJson, _ := json.Marshal(object)
    fmt.Printf("%s\n", oJson)
}
$ {"Number":5,"Word":"Packt"}
```

Conveniently, it also works pretty well with structures but what if I want to not use uppercase in the JSON data? You can define the output/input name of the JSON in the structure declaration:

```
type MyObject struct {
    Number int
    Word string
}
```

```
func main() {
    object := MyObject{5, "Packt"}
    oJson, _ := json.Marshal(object)
    fmt.Printf("%s\n", oJson)
}
$ {"number":5,"string":"Packt"}
```

We have not only lowercased the names of the keys, but we have even changed the name of the Word key to string.

Enough of marshalling, we will receive JSON data as an array of bytes, but the process is very similar with some changes:

```
type MyObject struct {
    Number int `json:"number"`
    Word string `json:"string"`
}

func main() {
    jsonBytes := []byte(`{"number":5, "string":"Packt"}`)
    var object MyObject
    err := json.Unmarshal(jsonBytes, &object)
    if err != nil {
        panic(err)
    }
    fmt.Printf("Number is %d, Word is %s\n", object.Number, object.Word)
}
```

The big difference here is that you have to allocate the space for the structure first (with a zero value) and then pass the reference to the method Unmarshal so that it tries to fill it. When you use Unmarshal, the first parameter is the array of bytes that contains the JSON information while the second parameter is the reference (that's why we are using an ampersand) to the structure we want to fill. Finally, let's use a generic map[string]interface{} method to hold the content of a JSON:

```
type MyObject struct {
    Number int      `json:"number"`
    Word string    `json:"string"`
}

func main() {
    jsonBytes := []byte(`{"number":5, "string":"Packt"}`)
    var dangerousObject map[string]interface{}
    err := json.Unmarshal(jsonBytes, &dangerousObject)
    if err != nil {
        panic(err)
    }
}
```

```
fmt.Printf("Number is %d, ", dangerousObject["number"])
fmt.Printf("Word is %s\n", dangerousObject["string"])
fmt.Printf("Error reference is %v\n",
dangerousObject["nothing"])
}
$ Number is %!d(float64=5), Word is Packt
Error reference is <nil>
```

What happened in the result? This is why we described the object as dangerous. You can point to a `nil` location when using this mode if you call a non-existing key in the JSON. Not only this, like in the example, it could also interpret a value as a `float64` when it is simply a `byte`, wasting a lot of memory.

So remember to just use `map[string]interface{}` when you need dirty quick access to JSON data that is fairly simple and you have under control the type of scenarios described previously.

Go tools

Go comes with a series of useful tools to ease the development process every day. Also in the golang page of GitHub, there are some tools that are supported by the Go team but they are not part of the compiler.

Most of the projects use tools such as `gofmt` so that all the code base looks similar. Godoc helps us to find useful information in Go's documentation and the `goimport` command to auto-import the packages we are using. Let's see them.

The golint tool

A linter analyzes source code to detect errors or improvements. The `golint` linter is available on <https://github.com/golang/lint> for installation (it doesn't come bundled with the compiler). It is very easy to use and is integrated some IDEs to be run when you save a source code file (Atom or Sublime Text, for example). Do you remember the implicit/explicit code that we run when talking about variables? Let's lint it:

```
//Explicitly declaring a "string" variable
var explicit string = "Hello, I'm a explicitly declared variable"

//Implicitly declaring a "string".
Type inferred inferred := "", I'm an inferred variable "

$ golint main.go
```

The `main.go:10:21:` command should omit the type string from the declaration of the `explicitString` variable; it will be inferred from the right-hand side.

It is telling us that Go compiler will actually infer this type of a variable from the code and you don't need to declare its type. What about the `Train` type on the interface section?

```
Type Train struct {
    TrainWidth int
}

$ golint main.go
```

The `main.go:5:6:` type exported `Train` type should have a comment or remain not exported.

In this case, it's pointing us that a public type such as `Train` type must be commented so that users can read the generated documentation to know its behavior.

The `gofmt` tool

The `gofmt` tool comes bundled with the compiler that already has access to it. Its purpose is to provide a set of indentation, formatting, spacing and few other rules to achieve good-looking Go code. For example, let's take the code of Hello World and make it a bit weirder by inserting spaces everywhere:

```
package main

func main(){
    println("Hello World!")
}

$ gofmt main.go
package main

func main() {
    println("Hello World!")
}
```

The `gofmt` command prints it correctly again. What is more, we can use the `-w` flag to overwrite the original file:

```
$ gofmt -w main.go
```

And now we'll have our file properly corrected.

The godoc tool

Go documentation is pretty extended and verbose. You can find detailed information about any topic you want to achieve. The `godoc` tool also helps you access this documentation directly from the command line. For example, we can query the package `encoding/json`:

```
$ godoc cmd/encoding/json
[...]
FUNCTIONS

func Compact(dst *bytes.Buffer, src []byte) error
Compact appends to dst the JSON-encoded src with insignificant space
characters elided.

func HTMLEscape(dst *bytes.Buffer, src []byte)
[...]
```

You can also use `grep`, a bash utility for Linux and Mac, to find specific information about some functionality. For example, we'll use `grep` to look for text that mentions anything about parsing JSON files:

```
$ godoc cmd/encoding/json | grep parse
```

The `Unmarshal` command parses the JSON encoded data and stores the result in the object being parsed.

One of the things that the `golint` command warns about is to use the beginning of a comment with the same name of the function it describes. This way, if you don't remember the name of the function that parses JSON, you can use `godoc` with `grep` and search for `parse` so the beginning of the line will always be the function name like in the example preceding the `Unmarshal` command.

The goimport tool

The `goimport` tool is a must have in Go. Sometimes you remember your packages so well that you don't need to search much to remember their API but it's more difficult to remember the project they belong to when doing the import. The `goimport` command helps you by searching your `$GOPATH` for occurrences of a package that you could be using to provide you with the project `import` line automatically. This is very useful if you configure your IDE to run `goimport` on save so that all used packages in the source file are imported automatically if you used them. It also works the other way around--if you delete the function you were using from a package and the package isn't being used anymore, it will remove the `import` line.

Contributing to Go open source projects in GitHub

One important thing to mention about Go packaging system is that it needs to have a proper folder structure within the GOPATH. This introduces a small problem when working with GitHub projects. We are used to forking a project, cloning our fork and start working before committing the pull-request to the original project. Wrong!

When you fork a project, you create a new repository on GitHub within your username. If you clone this repository and start working with it, all new import references in the project will point to your repository instead of the original! Imagine the following case in the original repository:

```
package main
import "github.com/original/a_library"
[some code]
```

Then, you make a fork and add a subfolder with a library called `a_library/my_library` that you want to use from the main package. The result is going to be the following:

```
package main
import (
    "github.com/original/a_library"
    "github.com/myaccount/a_library/my_library"
)
```

Now if you commit this line, the original repository that contains the code you have pushed will download this code anyways from your account again and it will use the references downloaded! Not the ones contained in the project!

So, the solution to this is simply to replace the `git clone` command with a `go get` pointing to the original library:

```
$ go get github.com/original/a_library
$ cd $GOPATH/src/github.com/original/a_library
$ git remote add my_origin https://github.com/myaccount/a_library
```

With this modification, you can work normally in the original code without fear as the references will stay correct. Once you are done you just have to commit and push to your remote.

```
$ git push my_origin my_branch
```

This way, you can now access the GitHub web user interface and open the pull request without polluting the actual original code with references to your account.

Summary

After this first chapter, you must be familiar with the syntax of Go and some of the command-line tools that come bundled with the compiler. We have left apart concurrency capabilities for a later chapter as they are large and pretty complex to grasp at the beginning so that the reader learns the syntax of the language first, becomes familiar and confident with it, and then they can jump to understanding **Communicating Sequential Processes (CSP)** concurrency patterns and distributed applications. The next steps are to start with the creational design patterns.

2

Creational Patterns - Singleton, Builder, Factory, Prototype, and Abstract Factory Design Patterns

We have defined two types of cars-luxury and family. The car Factory will have to return The first groups of design patterns that we are going to cover are the Creational patterns. As the name implies, it groups common practices for creating objects, so object creation is more encapsulated from the users that need those objects. Mainly, creational patterns try to give ready-to-use objects to users instead of asking for their creation, which, in some cases, could be complex, or which would couple your code with the concrete implementations of the functionality that should be defined in an interface.

Singleton design pattern - having a unique instance of a type in the entire program

Have you ever done interviews for software engineers? It's interesting that when you ask them about design patterns, more than 80% will mention **Singleton** design pattern. Why is that? Maybe it's because it is one of the most used design patterns out there or one of the easiest to grasp. We will start our journey on creational design patterns because of the latter reason.

Description

The Singleton pattern is easy to remember. As the name implies, it will provide you with a single instance of an object, and guarantee that there are no duplicates.

At the first call to use the instance, it is created and then reused between all the parts in the application that need to use that particular behavior.

You'll use the Singleton pattern in many different situations. For example:

- When you want to use the same connection to a database to make every query
- When you open a **Secure Shell (SSH)** connection to a server to do a few tasks, and don't want to reopen the connection for each task
- If you need to limit the access to some variable or space, you use a Singleton as the door to this variable (we'll see in the following chapters that this is more achievable in Go using channels anyway)
- If you need to limit the number of calls to some places, you create a Singleton instance to make the calls in the accepted window

The possibilities are endless, and we have just mentioned some of them.

Objectives

As a general guide, we consider using the Singleton pattern when the following rule applies:

- We need a single, shared value, of some particular type.
- We need to restrict object creation of some type to a single unit along the entire program.

Example - a unique counter

As an example of an object of which we must ensure that there is only one instance, we will write a counter that holds the number of times it has been called during program execution. It shouldn't matter how many instances we have of the counter, all of them must *count* the same value and it must be consistent between the instances.

Requirements and acceptance criteria

There are some requirements and acceptance criteria to write the described single counter.. They are as follows:

- When no counter has been created before, a new one is created with the value 0
- If a counter has already been created, return this instance that holds the actual count
- If we call the method `AddOne`, the count must be incremented by 1

We have a scenario with three tests to check in our unit tests.

Writing unit tests first

Go's implementation of this pattern is slightly different from what you'll find in pure object-oriented languages such as Java or C++, where you have static members. In Go, there's nothing like static members, but we have package scope to deliver a similar result.

To set up our project, we must create a new folder within our `$GOPATH/src` directory. The general rule as we mentioned in the [Chapter 1, Ready... Steady... Go!](#), is to create a subfolder with the VCS provider (such as GitHub), the username, and the name of the project.

For example, in my case, I use GitHub as my VCS and my username is `sayden`, so I will create the path `$GOPATH/src/github.com/sayden/go-design-patterns/creational/singleton`. The `go-design-patterns` instance in the path is the project name, the `creational` subfolder will also be our library name, and `singleton` the name of this particular package and subfolder:

```
mkdir -p $GOPATH/src/github.com/sayden/go-design-patterns/creational/singleton  
cd $GOPATH/src/github.com/sayden/go-design-patterns/creational/singleton
```

Create a new file inside the `singleton` folder called `singleton.go` to also reflect the name of the package and write the following package declarations for the `singleton` type:

```
package singleton  
  
type Singleton interface {  
    AddOne() int  
}  
  
type singleton struct {
```

```
    count int
}

var instance *singleton

func GetInstance() Singleton {
    return nil
}
func (s *singleton) AddOne() int {
    return 0
}
```

As we are following a TDD approach while writing the code, let's code the tests that use the functions we have just declared. The tests are going to be defined by following the acceptance criteria that we have written earlier. By convention in test files, we must create a file with the same name as the file to test, suffixed with `_test.go`. Both must reside in the same folder:

```
package singleton

import "testing"

func TestGetInstance(t *testing.T) {
    counter1 := GetInstance()

    if counter1 == nil {
        //Test of acceptance criteria 1 failed
        t.Error("expected pointer to Singleton after calling
GetInstance(), not nil")
    }

    expectedCounter := counter1
}
```

The first test checks something obvious, but no less important, in complex applications. We actually receive something when we ask for an instance of the counter. We have to think of it as a Creational pattern--we delegate the creation of the object to an unknown package that could fail in the creation or retrieval of the object. We also store the current counter in the `expectedCounter` variable to make a comparison later:

```
currentCount := counter1.AddOne()
if currentCount != 1 {
    t.Errorf("After calling for the first time to count, the count must be
1 but it is %d\n", currentCount)
}
```

Now we take advantage of the zero-initialization feature of Go. Remember that integer types in Go cannot be nil and as we know, that this is the first call to the counter, and it is an integer type of variable, and we also know that it is zero-initialized. So after the first call to the `AddOne()` function, the value of the count must be 1.

The test that checks the second condition proves that the `expectedConnection` variable is not different to the returned connection that we requested later. If they were different, the message `Singleton instances must be different` will cause the test to fail:

```
counter2 := GetInstance()
if counter2 != expectedCounter {
    //Test 2 failed
    t.Error("Expected same instance in counter2 but it got a different
instance")
}
```

The last test is simply counting 1 again with the second instance. The previous result was 1, so now it must give us 2:

```
currentCount = counter2.AddOne()
if currentCount != 2 {
    t.Errorf("After calling 'AddOne' using the second counter, the current
count must be 2 but was %d\n", currentCount)
}
```

The last thing we have to do to finish our test part is to execute the tests to make sure that they are failing before implementation. If one of them doesn't fail, it implies that we have done something wrong, and we have to reconsider that particular test. We must open the terminal and navigate to the path of the singleton package to execute:

```
$ go test -v .
==== RUN TestGetInstance
--- FAIL: TestGetInstance (0.00s)
    singleton_test.go:9: expected pointer to Singleton after calling
GetInstance(), not nil
    singleton_test.go:15: After calling for the first time to count,
the count must be 1 but it is 0
    singleton_test.go:27: After calling 'AddOne' using the second
counter, the current count must be 2 but was 0
FAIL
exit status 1
FAIL
```

Implementation

Finally, we have to implement the Singleton pattern. As we mentioned earlier, we'll usually write a static method and instance to retrieve the Singleton instance in languages such as Java or C++. In Go, we don't have the keyword `static`, but we can achieve the same result by using the scope of the package. First, we create a struct that contains the object which we want to guarantee to be a Singleton during the execution of the program:

```
package creational

type singleton struct{
    count int
}

var instance *singleton

func GetInstance() *singleton {
    if instance == nil {
        instance = new(singleton)
    }
    return instance
}

func (s *singleton) AddOne() int {
    s.count++
    return s.count
}
```

We must pay close attention to this piece of code. In languages such as Java or C++, the variable `instance` would be initialized to `NULL` at the beginning of the program. In Go, you can initialize a pointer to a struct as `nil`, but you cannot initialize a structure to `nil` (the equivalent of `NULL`). So the `var instance *singleton` line defines a pointer to a struct of type `Singleton` as `nil`, and the variable called `instance`.

We created a `GetInstance` method that checks if the `instance` has not been initialized already (`instance == nil`), and creates an instance in the space already allocated in the line `instance = new(singleton)`. Remember, when we use the keyword `new`, we are creating a pointer to an instance of the type between the parentheses.

The `AddOne` method will take the count of the variable `instance`, raise it by 1, and return the current value of the counter.

Let's run now our unit tests again:

```
$ go test -v -run=GetInstance
==== RUN    TestGetInstance
--- PASS: TestGetInstance (0.00s)
PASS
ok
```

A few words about the Singleton design pattern

We have seen a very simple example of the Singleton pattern, partially applied to some situation, that is, a simple counter. Just keep in mind that the Singleton pattern will give you the power to have a unique instance of some struct in your application and that no package can create any clone of this struct.

With Singleton, you are also hiding the complexity of creating the object, in case it requires some computation, and the pitfall of creating it every time you need an instance of it if all of them are similar. All this code writing, checking if the variable already exists, and storage, are encapsulated in the singleton and you won't need to repeat it everywhere if you use a global variable.

Here we are learning the classic singleton implementation for single threaded context. We will see a concurrent singleton implementation when we reach the chapters about concurrency because this implementation is not thread safe!

Builder design pattern - reusing an algorithm to create many implementations of an interface

Talking about **Creational** design patterns, it looks pretty semantic to have a **Builder** design pattern. The Builder pattern helps us construct complex objects without directly instantiating their struct, or writing the logic they require. Imagine an object that could have dozens of fields that are more complex structs themselves. Now imagine that you have many objects with these characteristics, and you could have more. We don't want to write the logic to create all these objects in the package that just needs to use the objects.

Description

Instance creation can be as simple as providing the opening and closing braces {} and leaving the instance with zero values, or as complex as an object that needs to make some API calls, check states, and create objects for its fields. You could also have an object that is composed of many objects, something that's really idiomatic in Go, as it doesn't support inheritance.

At the same time, you could be using the same technique to create many types of objects. For example, you'll use almost the same technique to build a car as you would build a bus, except that they'll be of different sizes and number of seats, so why don't we reuse the construction process? This is where the Builder pattern comes to the rescue.

Objectives

A Builder design pattern tries to:

- Abstract complex creations so that object creation is separated from the object user
- Create an object step by step by filling its fields and creating the embedded objects
- Reuse the object creation algorithm between many objects

Example - vehicle manufacturing

The Builder design pattern has been commonly described as the relationship between a director, a few Builders, and the product they build. Continuing with our example of the car, we'll create a vehicle Builder. The process (widely described as the algorithm) of creating a vehicle (the product) is more or less the same for every kind of vehicle--choose vehicle type, assemble the structure, place the wheels, and place the seats. If you think about it, you could build a car and a motorbike (two Builders) with this description, so we are reusing the description to create cars in manufacturing. The director is represented by the `ManufacturingDirector` type in our example.

Requirements and acceptance criteria

As far as we have described, we must dispose of a Builder of type Car and Motorbike and a unique director called ManufacturingDirector to take builders and construct products. So the requirements for a Vehicle builder example would be the following:

- I must have a manufacturing type that constructs everything that a vehicle needs
- When using a car builder, the VehicleProduct with four wheels, five seats, and a structure defined as Car must be returned
- When using a motorbike builder, the VehicleProduct with two wheels, two seats, and a structure defined as Motorbike must be returned
- A VehicleProduct built by any BuildProcess builder must be open to modifications

Unit test for the vehicle builder

With the previous acceptance criteria, we will create a director variable, the ManufacturingDirector type, to use the build processes represented by the product builder variables for a car and motorbike. The director is the one in charge of construction of the objects, but the builders are the ones that return the actual vehicle. So our builder declaration will look as follows:

```
package creational

type BuildProcess interface {
    SetWheels() BuildProcess
    SetSeats() BuildProcess
    SetStructure() BuildProcess
    GetVehicle() VehicleProduct
}
```

This preceding interface defines the steps that are necessary to build a vehicle. Every builder must implement this interface if they are to be used by the manufacturing. On every Set step, we return the same build process, so we can chain various steps together in the same statement, as we'll see later. Finally, we'll need a GetVehicle method to retrieve the Vehicle instance from the builder:

```
type ManufacturingDirector struct {}

func (f *ManufacturingDirector) Construct() {
    //Implementation goes here
}
```

```
func (f *ManufacturingDirector) SetBuilder(b BuildProcess) {
    //Implementation goes here
}
```

The `ManufacturingDirector` director variable is the one in charge of accepting the builders. It has a `Construct` method that will use the builder that is stored in `Manufacturing`, and will reproduce the required steps. The `SetBuilder` method will allow us to change the builder that is being used in the `Manufacturing` director:

```
type VehicleProduct struct {
    Wheels     int
    Seats      int
    Structure  string
}
```

The product is the final object that we want to retrieve while using the manufacturing. In this case, a vehicle is composed of wheels, seats, and a structure:

```
type CarBuilder struct {}

func (c *CarBuilder) SetWheels() BuildProcess {
    return nil
}

func (c *CarBuilder) SetSeats() BuildProcess {
    return nil
}

func (c *CarBuilder) SetStructure() BuildProcess {
    return nil
}

func (c *CarBuilder) Build() VehicleProduct {
    return VehicleProduct{}
}
```

The first Builder is the `Car` builder. It must implement every method defined in the `BuildProcess` interface. This is where we'll set the information for this particular builder:

```
type BikeBuilder struct {}

func (b *BikeBuilder) SetWheels() BuildProcess {
    return nil
}

func (b *BikeBuilder) SetSeats() BuildProcess {
    return nil
}
```

```
}

func (b *BikeBuilder) SetStructure() BuildProcess {
    return nil
}

func (b *BikeBuilder) Build() VehicleProduct {
    return VehicleProduct{}
}
```

The Motorbike structure must be the same as the Car structure, as they are all Builder implementations, but keep in mind that the process of building each can be very different. With this declaration of objects, we can create the following tests:

```
package creational

import "testing"

func TestBuilderPattern(t *testing.T) {
    manufacturingComplex := ManufacturingDirector{}

    carBuilder := &CarBuilder{}
    manufacturingComplex.SetBuilder(carBuilder)
    manufacturingComplex.Construct()

    car := carBuilder.Build()

    //code continues here...
```

We will start with the Manufacturing director and the Car Builder to fulfill the first two acceptance criteria. In the preceding code, we are creating our Manufacturing director that will be in charge of the creation of every vehicle during the test. After creating the Manufacturing director, we created a CarBuilder that we then passed to manufacturing by using the SetBuilder method. Once the Manufacturing director knows what it has to construct now, we can call the Construct method to create the VehicleProduct using CarBuilder. Finally, once we have all the pieces for our car, we call the GetVehicle method on CarBuilder to retrieve a Car instance:

```
if car.Wheels != 4 {
    t.Errorf("Wheels on a car must be 4 and they were %d\n", car.Wheels)
}

if car.Structure != "Car" {
    t.Errorf("Structure on a car must be 'Car' and was %s\n",
    car.Structure)
}
```

```
if car.Seats != 5 {  
    t.Errorf("Seats on a car must be 5 and they were %d\n", car.Seats)  
}
```

We have written three small tests to check if the outcome is a car. We checked that the car has four wheels, the structure has the description `Car`, and the number of seats is five. We have enough data to execute the tests and make sure that they are failing so that we can consider them reliable:

```
$ go test -v -run=TestBuilder .  
==== RUN TestBuilderPattern  
--- FAIL: TestBuilderPattern (0.00s)  
    builder_test.go:15: Wheels on a car must be 4 and they were 0  
    builder_test.go:19: Structure on a car must be 'Car' and was  
    builder_test.go:23: Seats on a car must be 5 and they were 0  
FAIL
```

Perfect! Now we will create tests for a Motorbike builder that covers the third and fourth acceptance criteria:

```
bikeBuilder := &BikeBuilder{ }  
  
manufacturingComplex.SetBuilder(bikeBuilder)  
manufacturingComplex.Construct()  
  
motorbike := bikeBuilder.GetVehicle()  
motorbike.Seats = 1  
  
if motorbike.Wheels != 2 {  
    t.Errorf("Wheels on a motorbike must be 2 and they were %d\n",  
    motorbike.Wheels)  
}  
  
if motorbike.Structure != "Motorbike" {  
    t.Errorf("Structure on a motorbike must be 'Motorbike' and was %s\n",  
    motorbike.Structure)  
}
```

The preceding code is a continuation of the car tests. As you can see, we reuse the previously created manufacturing to create the bike now by passing the `Motorbike` builder to it. Then we hit the `construct` button again to create the necessary parts, and call the `builder GetVehicle` method to retrieve the motorbike instance.

Take a quick look, because we have changed the default number of seats for this particular motorbike to 1. What we want to show here is that even while having a builder, you must also be able to change the default information in the returned instance to fit some specific needs. As we set the wheels manually, we won't test this feature.

Re-running the tests triggers the expected behavior:

```
$ go test -v -run=Builder .
==== RUN    TestBuilderPattern
--- FAIL: TestBuilderPattern (0.00s)
        builder_test.go:15: Wheels on a car must be 4 and they were 0
        builder_test.go:19: Structure on a car must be 'Car' and was
        builder_test.go:23: Seats on a car must be 5 and they were 0
        builder_test.go:35: Wheels on a motorbike must be 2 and they were 0
        builder_test.go:39: Structure on a motorbike must be 'Motorbike'
        and was
FAIL
```

Implementation

We will start implementing the manufacturing. As we said earlier (and as we set in our unit tests), the Manufacturing director must accept a builder and construct a vehicle using the provided builder. To recall, the BuildProcess interface will define the common steps needed to construct any vehicle and the Manufacturing director must accept builders and construct vehicles together with them:

```
package creational

type ManufacturingDirector struct {
    builder BuildProcess
}

func (f *ManufacturingDirector) SetBuilder(b BuildProcess) {
    f.builder = b
}

func (f *ManufacturingDirector) Construct() {
    f.builder.SetSeats().SetStructure().SetWheels()
}
```

Our ManufacturingDirector needs a field to store the builder in use; this field will be called `builder`. The `SetBuilder` method will replace the stored builder with the one provided in the arguments. Finally, take a closer look at the `Construct` method. It takes the builder that has been stored and reproduces the `BuildProcess` method that will create a full vehicle of some unknown type. As you can see, we have used all the setting calls in the same line thanks to returning the `BuildProcess` interface on each of the calls. This way the code is more compact:



Have you realized that the director entity in the Builder pattern is a clear candidate for a Singleton pattern too? In some scenarios, it could be critical that just an instance of the Director is available, and that is where you'll create a Singleton pattern for the Director of the Builder only. Design patterns composition is a very common technique and a very powerful one!

```
type CarBuilder struct {
    v VehicleProduct
}

func (c *CarBuilder) SetWheels() BuildProcess {
    c.v.Wheels = 4
    return c
}

func (c *CarBuilder) SetSeats() BuildProcess {
    c.v.Seats = 5
    return c
}

func (c *CarBuilder) SetStructure() BuildProcess {
    c.v.Structure = "Car"
    return c
}

func (c *CarBuilder) GetVehicle() VehicleProduct {
    return c.v
}
```

Here is our first builder, the car builder. A builder will need to store a `VehicleProduct` object, which here we have named `v`. Then we set the specific needs that a car has in our business--four wheels, five seats, and a structure defined as `Car`. In the `GetVehicle` method, we just return the `VehicleProduct` stored within the Builder that must be already constructed by the `ManufacturingDirector` type.

```
type BikeBuilder struct {
```

```
v VehicleProduct  
}  
  
func (b *BikeBuilder) SetWheels() BuildProcess {  
    b.v.Wheels = 2  
    return b  
}  
  
func (b *BikeBuilder) SetSeats() BuildProcess {  
    b.v.Seats = 2  
    return b  
}  
  
func (b *BikeBuilder) SetStructure() BuildProcess {  
    b.v.Structure = "Motorbike"  
    return b  
}  
  
func (b *BikeBuilder) GetVehicle() VehicleProduct {  
    return b.v  
}
```

The Motorbike Builder is the same as the car builder. We defined a motorbike to have two wheels, two seats, and a structure called Motorbike. It's very similar to the car object, but imagine that you want to differentiate between a sports motorbike (with only one seat) and a cruise motorbike (with two seats). You could simply create a new structure for sports motorbikes that implements the build process.

You can see that it's a repetitive pattern, but within the scope of every method of the BuildProcess interface, you could encapsulate as much complexity as you want such that the user need not know the details about the object creation.

With the definition of all the objects, let's run the tests again:

```
==== RUN  TestBuilderPattern  
--- PASS: TestBuilderPattern (0.00s)  
PASS  
ok  _/home/mcastro/pers/go-design-patterns/creational 0.001s
```

Well done! Think how easy it could be to add new vehicles to the ManufacturingDirector director just create a new class encapsulating the data for the new vehicle. For example, let's add a BusBuilder struct:

```
type BusBuilder struct {  
    v VehicleProduct  
}
```

```
func (b *BusBuilder) SetWheels() BuildProcess {
    b.v.Wheels = 4*2
    return b
}

func (b *BusBuilder) SetSeats() BuildProcess {
    b.v.Seats = 30
    return b
}

func (b *BusBuilder) SetStructure() BuildProcess {
    b.v.Structure = "Bus"
    return b
}

func (b *BusBuilder) GetVehicle() VehicleProduct {
    return b.v
}
```

That's all; your `ManufacturingDirector` would be ready to use the new product by following the `Builder` design pattern.

Wrapping up the `Builder` design pattern

The `Builder` design pattern helps us maintain an unpredictable number of products by using a common construction algorithm that is used by the director. The construction process is always abstracted from the user of the product.

At the same time, having a defined construction pattern helps when a newcomer to our source code needs to add a new product to the *pipeline*. The `BuildProcess` interface specifies what he must comply to be part of the possible builders.

However, try to avoid the `Builder` pattern when you are not completely sure that the algorithm is going to be more or less stable because any small change in this interface will affect all your builders and it could be awkward if you add a new method that some of your builders need and others Builders do not.

Factory method - delegating the creation of different types of payments

The Factory method pattern (or simply, Factory) is probably the second-best known and used design pattern in the industry. Its purpose is to abstract the user from the knowledge of the struct he needs to achieve for a specific purpose, such as retrieving some value, maybe from a web service or a database. The user only needs an interface that provides him this value. By delegating this decision to a Factory, this Factory can provide an interface that fits the user needs. It also eases the process of downgrading or upgrading of the implementation of the underlying type if needed.

Description

When using the Factory method design pattern, we gain an extra layer of encapsulation so that our program can grow in a controlled environment. With the Factory method, we delegate the creation of families of objects to a different package or object to abstract us from the knowledge of the pool of possible objects we could use. Imagine that you want to organize your holidays using a trip agency. You don't deal with hotels and traveling and you just tell the agency the destination you are interested in so that they provide you with everything you need. The trip agency represents a Factory of trips.

Objectives

After the previous description, the following objectives of the Factory Method design pattern must be clear to you:

- Delegating the creation of new instances of structures to a different part of the program
- Working at the interface level instead of with concrete implementations
- Grouping families of objects to obtain a family object creator

The example - a factory of payment methods for a shop

For our example, we are going to implement a payments method Factory, which is going to provide us with different ways of paying at a shop. In the beginning, we will have two methods of paying--cash and credit card. We'll also have an interface with the method, `Pay`, which every struct that wants to be used as a payment method must implement.

Acceptance criteria

Using the previous description, the requirements for the acceptance criteria are the following:

- To have a common method for every payment method called `Pay`
- To be able to delegate the creation of payments methods to the Factory
- To be able to add more payment methods to the library by just adding it to the factory method

First unit test

A Factory method has a very simple structure; we just need to identify how many implementations of our interface we are storing, and then provide a method, `GetPaymentMethod`, where you can pass a type of payment as an argument:

```
type PaymentMethod interface {
    Pay(amount float32) string
}
```

The preceding lines define the interface of the payment method. They define a way of making a payment at the shop. The Factory method will return instances of types that implement this interface:

```
const (
    Cash      = 1
    DebitCard = 2
)
```

We have to define the identified payment methods of the Factory as constants so that we can call and check the possible payment methods from outside of the package.

```
func GetPaymentMethod(m int) (PaymentMethod, error) {
```

```
        return nil, errors.New("Not implemented yet")
    }
```

The preceding code is the function that will create the objects for us. It returns a pointer, which must have an object that implements the `PaymentMethod` interface, and an error if asked for a method which is not registered.

```
type CashPM struct{}
type DebitCardPM struct{}

func (c *CashPM) Pay(amount float32) string {
    return ""
}

func (c *DebitCardPM) Pay(amount float32) string {
    return ""
}
```

To finish the declaration of the Factory, we create the two payment methods. As you can see, the `CashPM` and `DebitCardPM` structs implement the `PaymentMethod` interface by declaring a method, `Pay(amount float32) string`. The returned string will contain information about the payment.

With this declaration, we will start by writing the tests for the first acceptance criteria: to have a common method to retrieve objects that implement the `PaymentMethod` interface:

```
package creational

import (
    "strings"
    "testing"
)

func TestCreatePaymentMethodCash(t *testing.T) {
    payment, err := GetPaymentMethod(Cash)
    if err != nil {
        t.Fatal("A payment method of type 'Cash' must exist")
    }

    msg := payment.Pay(10.30)
    if !strings.Contains(msg, "paid using cash") {
        t.Error("The cash payment method message wasn't correct")
    }
    t.Log("LOG:", msg)
}
```

Now we'll have to separate the tests among a few of the test functions. `GetPaymentMethod` is a common method to retrieve methods of payment. We use the constant `Cash`, which we have defined in the implementation file (if we were using this constant outside for the scope of the package, we would call it using the name of the package as the prefix, so the syntax would be `creational.Cash`). We also check that we have not received an error when asking for a payment method. Observe that if we receive the error when asking for a payment method, we call `t.Fatal` to stop the execution of the tests; if we called just `t.Error` like in the previous tests, we would have a problem in the next lines when trying to access the `Pay` method of a nil object, and our tests would crash execution. We continue by using the `Pay` method of the interface by passing `10.30` as the amount. The returned message will have to contain the text `paid using cash`. The `t.Log(string)` method is a special method in testing. This struct allows us to write some logs when we run the tests if we pass the `-v` flag.

```
func TestGetPaymentMethodDebitCard(t *testing.T) {
    payment, err = GetPaymentMethod(Debit9Card)

    if err != nil {
        t.Error("A payment method of type 'DebitCard' must exist")
    }

    msg = payment.Pay(22.30)

    if !strings.Contains(msg, "paid using debit card") {
        t.Error("The debit card payment method message wasn't correct")
    }

    t.Log("LOG:", msg)
}
```

We repeat the same operation with the debit card method. We ask for the payment method defined with the constant `DebitCard`, and the returned message, when paying with debit card, must contain the `paid using debit card` string.

```
func TestGetPaymentMethodNonExistent(t *testing.T) {
    payment, err = GetPaymentMethod(20)

    if err == nil {
        t.Error("A payment method with ID 20 must return an error")
    }

    t.Log("LOG:", err)
}
```

Finally, we are going to test the situation when we request a payment method that doesn't exist (represented by the number 20, which doesn't match any recognized constant in the Factory). We will check if an error message (any) is returned when asking for an unknown payment method.

Let's check whether all tests are failing:

```
$ go test -v -run=GetPaymentMethod .
==== RUN    TestGetPaymentMethodCash
---- FAIL: TestGetPaymentMethodCash (0.00s)
        factory_test.go:11: A payment method of type 'Cash' must exist
==== RUN    TestGetPaymentMethodDebitCard
---- FAIL: TestGetPaymentMethodDebitCard (0.00s)
        factory_test.go:24: A payment method of type 'DebitCard' must exist
==== RUN    TestGetPaymentMethodNonExistent
---- PASS: TestGetPaymentMethodNonExistent (0.00s)
        factory_test.go:38: LOG: Not implemented yet
FAIL
exit status 1
FAIL
```

As you can see in this example, we can only see tests that return the `PaymentMethod` interfaces failing. In this case, we'll have to implement just a part of the code, and then test again before continuing.

Implementation

We will start with the `GetPaymentMethod` method. It must receive an integer that matches with one of the defined constants of the same file to know which implementation it should return.

```
package creational

import (
    "errors"
    "fmt"
)

type PaymentMethod interface {
    Pay(amount float32) string
}

const (
    Cash      = 1
    DebitCard = 2
```

```
)  
  
type CashPM struct{}  
type DebitCardPM struct{}  
  
func GetPaymentMethod(m int) (PaymentMethod, error) {  
    switch m {  
        case Cash:  
            return new(CashPM), nil  
        case DebitCard:  
            return new(DebitCardPM), nil  
        default:  
            return nil, errors.New(fmt.Sprintf("Payment method %d not  
recognized\n", m))  
    }  
}
```

We use a plain switch to check the contents of the argument `m` (method). If it matches any of the known methods--cash or debit card, it returns a new instance of them. Otherwise, it will return a nil and an error indicating that the payment method has not been recognized. Now we can run our tests again to check the second part of the unit tests:

```
$go test -v -run=GetPaymentMethod .  
==== RUN TestGetPaymentMethodCash  
---- FAIL: TestGetPaymentMethodCash (0.00s)  
    factory_test.go:16: The cash payment method message wasn't correct  
    factory_test.go:18: LOG:  
==== RUN TestGetPaymentMethodDebitCard  
---- FAIL: TestGetPaymentMethodDebitCard (0.00s)  
    factory_test.go:28: The debit card payment method message wasn't  
correct  
    factory_test.go:30: LOG:  
==== RUN TestGetPaymentMethodNonExistent  
---- PASS: TestGetPaymentMethodNonExistent (0.00s)  
    factory_test.go:38: LOG: Payment method 20 not recognized  
  
FAIL  
exit status 1  
FAIL
```

Now we do not get the errors saying it couldn't find the type of payment methods. Instead, we receive a message `not correct` error when it tries to use any of the methods that it covers. We also got rid of the `Not implemented` message that was being returned when we asked for an unknown payment method. Let's implement the structs now:

```
type CashPM struct{}  
type DebitCardPM struct{}  
  
func (c *CashPM) Pay(amount float32) string {  
    return fmt.Sprintf("%0.2f paid using cash\n", amount)  
}  
  
func (c *DebitCardPM) Pay(amount float32) string {  
    return fmt.Sprintf("%#0.2f paid using debit card\n", amount)  
}
```

We just get the amount, printing it in a nicely formatted message. With this implementation, the tests will all be passing now:

```
$ go test -v -run=GetPaymentMethod .  
==== RUN TestGetPaymentMethodCash  
--- PASS: TestGetPaymentMethodCash (0.00s)  
    factory_test.go:18: LOG: 10.30 paid using cash  
  
==== RUN TestGetPaymentMethodDebitCard  
--- PASS: TestGetPaymentMethodDebitCard (0.00s)  
    factory_test.go:30: LOG: 22.30 paid using debit card  
  
==== RUN TestGetPaymentMethodNonExistent  
--- PASS: TestGetPaymentMethodNonExistent (0.00s)  
    factory_test.go:38: LOG: Payment method 20 not recognized  
  
PASS  
ok
```

Do you see the `LOG:` messages? They aren't errors, we just print some information that we receive when using the package under test. These messages can be omitted unless you pass the `-v` flag to the test command:

```
$ go test -run=GetPaymentMethod .  
ok
```

Upgrading the Debitcard method to a new platform

Now imagine that your DebitCard payment method has changed for some reason, and you need a new struct for it. To achieve this scenario, you will only need to create the new struct and replace the old one when the user asks for the DebitCard payment method:

```
type CreditCardPM struct {}
func (d *CreditCardPM) Pay(amount float32) string {
    return fmt.Sprintf("%#0.2f paid using new credit card implementation\n",
amount)
}
```

This is our new type that will replace the DebitCardPM structure. The CreditCardPM implements the same PaymentMethod interface as the debit card. We haven't deleted the previous one in case we need it in the future. The only difference lies in the returned message that now contains the information about the new type. We also have to modify the method to retrieve the payment methods:

```
func GetPaymentMethod(m int) (PaymentMethod, error) {
    switch m {
        case Cash:
            return new(CashPM), nil
        case DebitCard:
            return new(CreditCardPM), nil
        default:
            return nil, errors.New(fmt.Sprintf("Payment method %d not
recognized\n", m))
    }
}
```

The only modification is in the line where we create the new debit card that now points to the newly created struct. Let's run the tests to see if everything is still correct:

```
$ go test -v -run=GetPaymentMethod .
==== RUN TestGetPaymentMethodCash
--- PASS: TestGetPaymentMethodCash (0.00s)
    factory_test.go:18: LOG: 10.30 paid using cash

==== RUN TestGetPaymentMethodDebitCard
--- FAIL: TestGetPaymentMethodDebitCard (0.00s)
    factory_test.go:28: The debit card payment method message wasn't
correct
    factory_test.go:30: LOG: 22.30 paid using new debit card
implementation
```

```
==== RUN    TestGetPaymentMethodNonExistent
--- PASS: TestGetPaymentMethodNonExistent (0.00s)
        factory_test.go:38: LOG: Payment method 20 not recognized

FAIL
exit status 1
FAIL
```

Uh, oh! Something has gone wrong. The expected message when paying with a credit card does not match the returned message. Does it mean that our code isn't correct? Generally speaking, yes, you shouldn't modify your tests to make your program work. When defining tests, you should be also aware of not defining them too much because you could achieve some coupling in the tests that you didn't have in your code. With the message restriction, we have a few grammatically correct possibilities for the message, so we'll change it to the following:

```
return fmt.Sprintf("%#0.2f paid using debit card (new)\n", amount)
```

We run the tests again now:

```
$ go test -v -run=GetPaymentMethod .
==== RUN    TestGetPaymentMethodCash
--- PASS: TestGetPaymentMethodCash (0.00s)
        factory_test.go:18: LOG: 10.30 paid using cash

==== RUN    TestGetPaymentMethodDebitCard
--- PASS: TestGetPaymentMethodDebitCard (0.00s)
        factory_test.go:30: LOG: 22.30 paid using debit card (new)

==== RUN    TestGetPaymentMethodNonExistent
--- PASS: TestGetPaymentMethodNonExistent (0.00s)
        factory_test.go:38: LOG: Payment method 20 not recognized

PASS
ok
```

Everything is okay again. This was just a small example of how to write good unit tests, too. When we wanted to check that a debit card payment method returns a message that contains paid using debit card string, we were probably being a bit restrictive, and it would be better to check for those words separately or define a better formatting for the returned messages.

What we learned about the Factory method

With the Factory method pattern, we have learned how to group families of objects so that their implementation is outside of our scope. We have also learned what to do when we need to upgrade an implementation of a used structs. Finally, we have seen that tests must be written with care if you don't want to tie yourself to certain implementations that don't have anything to do with the tests directly.

Abstract Factory - a factory of factories

After learning about the factory design pattern, where we grouped a family of related objects in our case payment methods, one can be quick to think--what if I group families of objects in a more structured hierarchy of families?

Description

The Abstract Factory design pattern is a new layer of grouping to achieve a bigger (and more complex) composite object, which is used through its interfaces. The idea behind grouping objects in families and grouping families is to have big factories that can be interchangeable and can grow more easily. In the early stages of development, it is also easier to work with factories and abstract factories than to wait until all concrete implementations are done to start your code. Also, you won't write an Abstract Factory from the beginning unless you know that your object's inventory for a particular field is going to be very large and it could be easily grouped into families.

The objectives

Grouping related families of objects is very convenient when your object number is growing so much that creating a unique point to get them all seems the only way to gain the flexibility of the runtime object creation. The following objectives of the Abstract Factory method must be clear to you:

- Provide a new layer of encapsulation for Factory methods that return a common interface for all factories
- Group common factories into a *super Factory* (also called a factory of factories)

The vehicle factory example, again?

For our example, we are going to reuse the Factory we created in the Builder design pattern. We want to show the similarities to solve the same problem using a different approach so that you can see the strengths and weaknesses of each approach. This is going to show you the power of implicit interfaces in Go, as we won't have to touch almost anything. Finally, we are going to create a new Factory to create shipment orders.

Acceptance criteria

The following are the acceptance criteria for using the `Vehicle` object's Factory method:

- We must retrieve a `Vehicle` object using a factory returned by the abstract factory.
- The vehicle must be a concrete implementation of a `Motorbike` or a `Car` that implements both interfaces (`Vehicle` and `Car` or `Vehicle` and `Motorbike`).

Unit test

This is going to be a long example, so pay attention, please. We will have the following entities:

- **Vehicle:** The interface that all objects in our factories must implement:
 - **Motorbike:** An interface for motorbikes of the types sport (one seat) and cruise (two seats).
 - **Car:** An interface for cars of types luxury (with four doors) and family (with five doors).
- **VehicleFactory:** An interface (the Abstract Factory) to retrieve factories that implement the `VehicleFactory` method:
 - **Motorbike Factory:** A factory that implements the `VehicleFactory` interface to return vehicle that implements the `Vehicle` and `Motorbike` interfaces.
 - **Car Factory:** Another factory that implements the `VehicleFactory` interface to return vehicles that implement the `Vehicle` and `Car` interfaces.

For clarity, we are going to separate each entity into a different file. We will start with the `Vehicle` interface, which will be in the `vehicle.go` file:

```
package abstract_factory

type Vehicle interface {
    NumWheels() int
    NumSeats() int
}
```

The `Car` and `Motorbike` interfaces will be in the `car.go` and `motorbike.go` files, respectively:

```
// Package abstract_factory file: car.go
package abstract_factory

type Car interface {
    NumDoors() int
}

// Package abstract_factory file: motorbike.go
package abstract_factory

type Motorbike interface {
    GetMotorbikeType() int
}
```

We have one last interface, the one that each factory must implement. This will be in the `vehicle_factory.go` file:

```
package abstract_factory

type VehicleFactory interface {
    NewVehicle(v int) (Vehicle, error)
}
```

So, now we are going to declare the car factory. It must implement the `VehicleFactory` interface defined previously to return `Vehicles` instances:

```
const (
    LuxuryCarType = 1
    FamilyCarType = 2
)

type CarFactory struct{}
func (c *CarFactory) NewVehicle(v int) (Vehicle, error) {
    switch v {
        case LuxuryCarType:
            return new(LuxuryCar), nil
    }
}
```

```
        case FamilyCarType:
            return new(FamilyCar), nil
        default:
            return nil, errors.New(fmt.Sprintf("Vehicle of type %d not
recognized\n", v))
    }
}
```

We have defined two types of cars--luxury and family. The car Factory will have to return cars that implement the Car and the Vehicle interfaces, so we need two concrete implementations:

```
//luxury_car.go
package abstract_factory

type LuxuryCar struct{}

func (*LuxuryCar) NumDoors() int {
    return 4
}
func (*LuxuryCar) NumWheels() int {
    return 4
}
func (*LuxuryCar) NumSeats() int {
    return 5
}

package abstract_factory

type FamilyCar struct{}

func (*FamilyCar) NumDoors() int {
    return 5
}
func (*FamilyCar) NumWheels() int {
    return 4
}
func (*FamilyCar) NumSeats() int {
    return 5
}
```

That's all for cars. Now we need the motorbike factory, which, like the car factory, must implement the VehicleFactory interface:

```
const (
    SportMotorbikeType = 1
    CruiseMotorbikeType = 2
)
```

```
type MotorbikeFactory struct{ }

func (m *MotorbikeFactory) Build(v int) (Vehicle, error) {
    switch v {
        case SportMotorbikeType:
            return new(SportMotorbike), nil
        case CruiseMotorbikeType:
            return new(CruiseMotorbike), nil
        default:
            return nil, errors.New(fmt.Sprintf("Vehicle of type %d not
recognized\n", v))
    }
}
```

For the motorbike Factory, we have also defined two types of motorbikes using the `const` keywords: `SportMotorbikeType` and `CruiseMotorbikeType`. We will switch over the `v` argument in the `Build` method to know which type shall be returned. Let's write the two concrete motorbikes:

```
//sport_motorbike.go
package abstract_factory

type SportMotorbike struct{ }

func (s *SportMotorbike) NumWheels() int {
    return 2
}
func (s *SportMotorbike) NumSeats() int {
    return 1
}
func (s *SportMotorbike) GetMotorbikeType() int {
    return SportMotorbikeType
}

//cruise_motorbike.go
package abstract_factory

type CruiseMotorbike struct{ }

func (c *CruiseMotorbike) NumWheels() int {
    return 2
}
func (c *CruiseMotorbike) NumSeats() int {
    return 2
}
func (c *CruiseMotorbike) GetMotorbikeType() int {
    return CruiseMotorbikeType
}
```

To finish, we need the abstract factory itself, which we will put in the previously created `vehicle_factory.go` file:

```
package abstract_factory

import (
    "fmt"
    "errors"
)

type VehicleFactory interface {
    Build(v int) (Vehicle, error)
}

const (
    CarFactoryType = 1
    MotorbikeFactoryType = 2
)

func BuildFactory(f int) (VehicleFactory, error) {
    switch f {
    default:
        return nil, errors.New(fmt.Sprintf("Factory with id %d not
recognized\n", f))
    }
}
```

We are going to write enough tests to make a reliable check as the scope of the book doesn't cover 100% of the statements. It will be a good exercise for the reader to finish these tests. First, a motorbike Factory test:

```
package abstract_factory

import "testing"

func TestMotorbikeFactory(t *testing.T) {
    motorbikeF, err := BuildFactory(MotorbikeFactoryType)
    if err != nil {
        t.Fatal(err)
    }

    motorbikeVehicle, err := motorbikeF.Build(SportMotorbikeType)
    if err != nil {
        t.Fatal(err)
    }

    t.Logf("Motorbike vehicle has %d wheels\n",
    motorbikeVehicle.NumWheels())
```

```
sportBike, ok := motorbikeVehicle.(Motorbike)
if !ok {
    t.Fatal("Struct assertion has failed")
}
t.Logf("Sport motorbike has type %d\n", sportBike.GetMotorbikeType())
}
```

We use the package method, `BuildFactory`, to retrieve a motorbike Factory (passing the `MotorbikeFactory` ID in the parameters), and check if we get any error. Then, already with the motorbike factory, we ask for a vehicle of the type `SportMotorbikeType` and check for errors again. With the returned vehicle, we can ask for methods of the vehicle interface (`NumWheels` and `NumSeats`). We know that it is a motorbike, but we cannot ask for the type of motorbike without using the type assertion. We use the type assertion on the vehicle to retrieve the motorbike that the `motorbikeVehicle` represents in the code line `sportBike, found := motorbikeVehicle.(Motorbike)`, and we must check that the type we have received is correct.

Finally, now we have a motorbike instance, we can ask for the bike type by using the `GetMotorbikeType` method. Now we are going to write a test that checks the car factory in the same manner:

```
func TestCarFactory(t *testing.T) {
    carF, err := BuildFactory(CarFactoryType)
    if err != nil {
        t.Fatal(err)
    }

    carVehicle, err := carF.Build(LuxuryCarType)
    if err != nil {
        t.Fatal(err)
    }

    t.Logf("Car vehicle has %d seats\n", carVehicle.NumWheels())

    luxuryCar, ok := carVehicle.(Car)
    if !ok {
        t.Fatal("Struct assertion has failed")
    }
    t.Logf("Luxury car has %d doors.\n", luxuryCar.NumDoors())
}
```

Again, we use the `BuildFactory` method to retrieve a Car Factory by using the `CarFactoryType` in the parameters. With this factory, we want a car of the `Luxury` type so that it returns a `vehicle` instance. We again do the type assertion to point to a car instance so that we can ask for the number of doors using the `NumDoors` method.

Let's run the unit tests:

```
go test -v -run=Factory .
==== RUN TestMotorbikeFactory
--- FAIL: TestMotorbikeFactory (0.00s)
    vehicle_factory_test.go:8: Factory with id 2 not recognized

==== RUN TestCarFactory
--- FAIL: TestCarFactory (0.00s)
    vehicle_factory_test.go:28: Factory with id 1 not recognized

FAIL
exit status 1
FAIL
```

Done. It can't recognize any factory as their implementation is still not done.

Implementation

The implementation of every factory is already done for the sake of brevity. They are very similar to the Factory method with the only difference being that in the Factory method, we don't use an instance of the Factory method because we use the package functions directly. The implementation of the vehicle Factory is as follows:

```
func BuildFactory(f int) (VehicleFactory, error) {
    switch f {
        case CarFactoryType:
            return new(CarFactory), nil
        case MotorbikeFactoryType:
            return new(MotorbikeFactory), nil
        default:
            return nil, errors.New(fmt.Sprintf("Factory with id %d not
recognized\n", f))
    }
}
```

Like in any factory, we switched between the factory possibilities to return the one that was demanded. As we have already implemented all concrete vehicles, the tests must run too:

```
go test -v -run=Factory -cover .
==== RUN TestMotorbikeFactory
--- PASS: TestMotorbikeFactory (0.00s)
    vehicle_factory_test.go:16: Motorbike vehicle has 2 wheels
    vehicle_factory_test.go:22: Sport motorbike has type 1
==== RUN TestCarFactory
--- PASS: TestCarFactory (0.00s)
```

```
vehicle_factory_test.go:36: Car vehicle has 4 seats
vehicle_factory_test.go:42: Luxury car has 4 doors.
PASS
coverage: 45.8% of statements
ok
```

All of them passed. Take a close look and note that we have used the `-cover` flag when running the tests to return a coverage percentage of the package: 45.8%. What this tells us is that 45.8% of the lines are covered by the tests we have written, but 54.2% are still not under the tests. This is because we haven't covered the cruise motorbike and the family car with the tests. If you write those tests, the result should rise to around 70.8%.



Type assertion is also known as **casting** in other languages. When you have an interface instance, which is essentially a pointer to a struct, you just have access to the interface methods. With type assertion, you can tell the compiler the type of the pointed struct, so you can access the entire struct fields and methods.

A few lines about the Abstract Factory method

We have learned how to write a factory of factories that provides us with a very generic object of vehicle type. This pattern is commonly used in many applications and libraries, such as cross-platform GUI libraries. Think of a button, a generic object, and button factory that provides you with a factory for Microsoft Windows buttons while you have another factory for Mac OS X buttons. You don't want to deal with the implementation details of each platform, but you just want to implement the actions for some specific behavior raised by a button.

Also, we have seen the differences when approaching the same problem with two different solutions--the Abstract factory and the Builder pattern. As you have seen, with the Builder pattern, we had an unstructured list of objects (cars with motorbikes in the same factory). Also, we encouraged reusing the building algorithm in the Builder pattern. In the Abstract factory, we have a very structured list of vehicles (the factory for motorbikes and a factory for cars). We also didn't mix the creation of cars with motorbikes, providing more flexibility in the creation process. The Abstract factory and Builder patterns can both resolve the same problem, but your particular needs will help you find the slight differences that should lead you to take one solution or the other.

Prototype design pattern

The last pattern we will see in this chapter is the **Prototype** pattern. Like all creational patterns, this too comes in handy when creating objects, and it is very common to see the Prototype pattern surrounded by more patterns.

While with the Builder pattern, we are dealing with repetitive building algorithms and with the factories we are simplifying the creation of many types of objects; with the Prototype pattern, we will use an already created instance of some type to clone it and complete it with the particular needs of each context. Let's see it in detail.

Description

The aim of the Prototype pattern is to have an object or a set of objects that is already created at compilation time, but which you can clone as many times as you want at runtime. This is useful, for example, as a default template for a user who has just registered with your webpage or a default pricing plan in some service. The key difference between this and a Builder pattern is that objects are cloned for the user instead of building them at runtime. You can also build a cache-like solution, storing information using a prototype.

Objective

The main objective for the Prototype design pattern is to avoid repetitive object creation. Imagine a default object composed of dozens of fields and embedded types. We don't want to write everything needed by this type every time that we use the object, especially if we can mess it up by creating instances with different *foundations*:

- Maintain a set of objects that will be cloned to create new instances
- Provide a default value of some type to start working on top of it
- Free CPU of complex object initialization to take more memory resources

Example

We will build a small component of an imaginary customized shirts shop that will have a few shirts with their default colors and prices. Each shirt will also have a **Stock Keeping Unit (SKU)**, a system to identify items stored at a specific location) that will need an update.

Acceptance criteria

To achieve what is described in the example, we will use a prototype of shirts. Each time we need a new shirt we will take this prototype, clone it and work with it. In particular, those are the acceptance criteria for using the Prototype pattern design method in this example:

- To have a shirt-cloner object and interface to ask for different types of shirts (white, black, and blue at 15.00, 16.00, and 17.00 dollars respectively)
- When you ask for a white shirt, a clone of the white shirt must be made, and the new instance must be different from the original one
- The SKU of the created object shouldn't affect new object creation
- An info method must give me all the information available on the instance fields, including the updated SKU

Unit test

First, we will need a `ShirtCloner` interface and an object that implements it. Also, we need a package-level function called `GetShirtsCloner` to retrieve a new instance of the cloner:

```
type ShirtCloner interface {
    GetClone(s int) (ItemInfoGetter, error)
}

const (
    White = 1
    Black = 2
    Blue  = 3
)

func GetShirtsCloner() ShirtCloner {
    return nil
}

type ShirtsCache struct {}
func (s *ShirtsCache)GetClone(s int) (ItemInfoGetter, error) {
    return nil, errors.New("Not implemented yet")
}
```

Now we need an object struct to clone, which implements an interface to retrieve the information of its fields. We will call the object `Shirt` and the `ItemInfoGetter` interface:

```
type ItemInfoGetter interface {
    GetInfo() string
}

type ShirtColor byte

type Shirt struct {
    Price float32
    SKU   string
    Color ShirtColor
}
func (s *Shirt) GetInfo() string {
    return ""
}

func GetShirtsCloner() ShirtCloner {
    return nil
}

var whitePrototype *Shirt = &Shirt{
    Price: 15.00,
    SKU:   "empty",
    Color: White,
}

func (i *Shirt) GetPrice() float32 {
    return i.Price
}
```



Have you realized that the type called `ShirtColor` that we defined is just a `byte` type? Maybe you are wondering why we haven't simply used the `byte` type. We could, but this way we created an easily readable struct, which we can upgrade with some methods in the future if required. For example, we could write a `String()` method that returns the color in the string format (`White` for type 1, `Black` for type 2, and `Blue` for type 3).

With this code, we can already write our first tests:

```
func TestClone(t *testing.T) {
    shirtCache := GetShirtsCloner()
    if shirtCache == nil {
        t.Fatal("Received cache was nil")
    }

    item1, err := shirtCache.GetClone(White)
    if err != nil {
        t.Error(err)
    }

    //more code continues here...
```

We will cover the first case of our scenario, where we need a cloner object that we can use to ask for different shirt colors.

For the second case, we will take the original object (which we can access because we are in the scope of the package), and we will compare it with our `shirt1` instance.

```
if item1 == whitePrototype {
    t.Error("item1 cannot be equal to the white prototype");
}
```

Now, for the third case. First, we will type assert `item1` to a shirt so that we can set an SKU. We will create a second shirt, also white, and we will type assert it too to check that the SKUs are different:

```
shirt1, ok := item1.(*Shirt)
if !ok {
    t.Fatal("Type assertion for shirt1 couldn't be done successfully")
}
shirt1.SKU = "abbcc"

item2, err := shirtCache.GetClone(White)
if err != nil {
    t.Fatal(err)
}

shirt2, ok := item2.(*Shirt)
if !ok {
    t.Fatal("Type assertion for shirt2 couldn't be done successfully")
}

if shirt1.SKU == shirt2.SKU {
    t.Error("SKU's of shirt1 and shirt2 must be different")
}
```

```
if shirt1 == shirt2 {  
    t.Error("Shirt 1 cannot be equal to Shirt 2")  
}
```

Finally, for the fourth case, we log the info of the first and second shirts:

```
t.Logf("LOG: %s", shirt1.GetInfo())  
t.Logf("LOG: %s", shirt2.GetInfo())
```

We will be printing the memory positions of both shirts, so we make this assertion at a more physical level:

```
t.Logf("LOG: The memory positions of the shirts are different %p != %p  
\n\n", &shirt1, &shirt2)
```

Finally, we run the tests so we can check that it fails:

```
go test -run=TestClone .  
--- FAIL: TestClone (0.00s)  
prototype_test.go:10: Not implemented yet  
FAIL  
FAIL
```

We have to stop there so that the tests don't panic if we try to use a nil object that is returned by the GetShirtsCloner function.

Implementation

We will start with the `GetClone` method. This method should return an item of the specified type and we have three type: white, black and blue:

```
var whitePrototype *Shirt = &Shirt{  
    Price: 15.00,  
    SKU: "empty",  
    Color: White,  
}  
  
var blackPrototype *Shirt = &Shirt{  
    Price: 16.00,  
    SKU: "empty",  
    Color: Black,  
}  
  
var bluePrototype *Shirt = &Shirt{  
    Price: 17.00,  
    SKU: "empty",  
}
```

```
    Color: Blue,  
}
```

So now that we have the three prototypes to work over we can implement GetClone(s int) method:

```
type ShirtsCache struct {}  
func (s *ShirtsCache)GetClone(s int) (ItemInfoGetter, error) {  
    switch m {  
        case White:  
            newItem := *whitePrototype  
            return &newItem, nil  
        case Black:  
            newItem := *blackPrototype  
            return &newItem, nil  
        case Blue:  
            newItem := *bluePrototype  
            return &newItem, nil  
        default:  
            return nil, errors.New("Shirt model not recognized")  
    }  
}
```

The Shirt structure also needs a GetInfo implementation to print the contents of the instances.

```
type ShirtColor byte  
  
type Shirt struct {  
    Price float32  
    SKU   string  
    Color ShirtColor  
}  
  
func (s *Shirt) GetInfo() string {  
    return fmt.Sprintf("Shirt with SKU '%s' and Color id %d that costs  
%f\n", s.SKU, s.Color, s.Price)  
}
```

Finally, let's run the tests to see that everything is now working:

```
go test -run=TestClone -v .
==== RUN    TestClone
--- PASS: TestClone (0.00s)
prototype_test.go:41: LOG: Shirt with SKU 'abbcc' and Color id 1 that costs
15.000000
prototype_test.go:42: LOG: Shirt with SKU 'empty' and Color id 1 that costs
15.000000
prototype_test.go:44: LOG: The memory positions of the shirts are different
0xc42002c038 != 0xc42002c040

PASS
ok
```

In the log, (remember to set the `-v` flag when running the tests) you can check that `shirt1` and `shirt2` have different SKUs. Also, we can see the memory positions of both objects. Take into account that the positions shown on your computer will probably be different.

What we learned about the Prototype design pattern

The Prototype pattern is a powerful tool to build caches and default objects. You have probably realized too that some patterns can overlap a bit, but they have small differences that make them more appropriate in some cases and not so much in others.

Summary

We have seen the five main creational design patterns commonly used in the software industry. Their purpose is to abstract the user from the creation of objects for complexity or maintainability purposes. They have been the foundation of thousands of applications and libraries since the 1990s, and most of the software we use today has many of these creational patterns under the hood.

It's worth mentioning that these patterns are not thread-free. In a more advanced chapter, we will see concurrent programming in Go, and how to create some of the more critical design patterns using a concurrent approach.

3

Structural Patterns - Composite, Adapter, and Bridge Design Patterns

We are going to start our journey through the world of structural patterns. Structural patterns, as the name implies, help us to shape our applications with commonly used structures and relationships.

The Go language, by nature, encourages use of composition almost exclusively by its lack of inheritance. Because of this, we have been using the **Composite** design pattern extensively until now, so let's start by defining the Composite design pattern.

Composite design pattern

The Composite design pattern favors composition (commonly defined as a *has a* relationship) over inheritance (an *is a* relationship). The *composition over inheritance* approach has been a source of discussions among engineers since the nineties. We will learn how to create object structures by using a *has a* approach. All in all, Go doesn't have inheritance because it doesn't need it!

Description

In the Composite design pattern, you will create hierarchies and trees of objects. Objects have different objects with their own fields and methods inside them. This approach is very powerful and solves many problems of inheritance and multiple inheritances. For example, a typical inheritance problem is when you have an entity that inherits from two completely different classes, which have absolutely no relationship between them. Imagine an athlete who trains, and who is a swimmer who swims:

- The Athlete class has a Train() method
- The Swimmer class has a Swim() method

The Swimmer class inherits from the Athlete class, so it inherits its Train method and declares its own Swim method. You could also have a cyclist who is also an athlete, and declares a Ride method.

But now imagine an animal that eats, like a dog that also barks:

- The Cyclist class has a Ride() method
- The Animal class has Eat(), Dog(), and Bark() methods

Nothing fancy. You could also have a fish that is an animal, and yes, swims! So, how do you solve it? A fish cannot be a swimmer that also trains. Fish don't train (as far as I know!). You could make a Swimmer interface with a Swim method, and make the swimmer athlete and fish implement it. This would be the best approach, but you still would have to implement swim method twice, so code reusability would be affected. What about a triathlete? They are athletes who swim, run, and ride. With multiple inheritances, you could have a sort of solution, but that will become complex and not maintainable very soon.

Objectives

As you have probably imagined already, the objective of the composition is to avoid this type of hierarchy hell where the complexity of an application could grow too much, and the clarity of the code is affected.

The swimmer and the fish

We will solve the described problem of the athlete and the fish that swims in a very idiomatic Go way. With Go, we can use two types of composition--the **direct** composition and the **embedding** composition. We will first solve this problem by using direct composition which is having everything that is needed as fields within the struct.

Requirements and acceptance criteria

Requirements are like the ones described previously. We'll have an athlete and a swimmer. We will also have an animal and a fish. The `Swimmer` and the `Fish` methods must share the code. The athlete must train, and the animal must eat:

- We must have an `Athlete` struct with a `Train` method
- We must have a `Swimmer` with a `Swim` method
- We must have an `Animal` struct with an `Eat` method
- We must have a `Fish` struct with a `Swim` method that is shared with the `Swimmer`, and not have inheritance or hierarchy issues

Creating compositions

The Composite design pattern is a pure structural pattern, and it doesn't have much to test apart from the structure itself. We won't write unit tests in this case, and we'll simply describe the ways to create those compositions in Go.

First, we'll start with the `Athlete` structure and its `Train` method:

```
type Athlete struct{}

func (a *Athlete) Train() {
    fmt.Println("Training")
}
```

The preceding code is pretty straightforward. Its `Train` method prints the word `Training` and a new line. We'll create a composite swimmer that has an `Athlete` struct inside it:

```
type CompositeSwimmerA struct{
    MyAthlete Athlete
    MySwim func()
}
```

The `CompositeSwimmerA` type has a `MyAthlete` field of type `Athlete`. It also stores a `func()` type. Remember that in Go, functions are first-class citizens and they can be used as parameters, fields, or arguments just like any variable. So `CompositeSwimmerA` has a `MySwim` field that stores a **closure**, which takes no arguments and returns nothing. How can I assign a function to it? Well, let's create a function that matches the `func()` signature (no arguments, no return):

```
func Swim() {
    fmt.Println("Swimming!")
}
```

That's all! The `Swim()` function takes no arguments and returns nothing, so it can be used as the `MySwim` field in the `CompositeSwimmerA` struct:

```
swimmer := CompositeSwimmerA{
    MySwim: Swim,
}

swimmer.MyAthlete.Train()
swimmer.MySwim()
```

Because we have a function called `Swim()`, we can assign it to the `MySwim` field. Note that the `Swim` type doesn't have the parenthesis that will execute its contents. This way we take the entire function and copy it to `MySwim` method.

But wait. We haven't passed any athlete to the `MyAthlete` field and we are using it! It's going to fail! Let's see what happens when we execute this snippet:

```
$ go run main.go
Training
Swimming!
```

That's weird, isn't it? Not really because of the nature of zero-initialization in Go. If you don't pass an `Athlete` struct to the `CompositeSwimmerA` type, the compiler will create one with its values zero-initialized, that is, an `Athlete` struct with its fields initialized to zero. Check out [Chapter 1, Ready... Steady... Go!](#) to recall zero-initialization if this seems confusing. Consider the `CompositeSwimmerA` struct code again:

```
type CompositeSwimmerA struct{
    MyAthlete Athlete
    MySwim     func()
}
```

Now we have a pointer to a function stored in the `MySwim` field. We can assign the `Swim` function the same way, but with an extra step:

```
localSwim := Swim

swimmer := CompositeSwimmerA{
    MySwim: localSwim,
}

swimmer.MyAthlete.Train()
swimmer.MySwim ()
```

First, we need a variable that contains the function `Swim`. This is because a function doesn't have an address to pass it to the `CompositeSwimmerA` type. Then, to use this function within the struct, we have to make a two-step call.

What about our fish problem? With our `Swim` function, it is not a problem anymore. First, we create the `Animal` struct:

```
type Animal struct{}

func (r *Animal) Eat() {
    println("Eating")
}
```

Then we'll create a `Shark` object that embeds the `Animal` object:

```
type Shark struct{
    Animal
    Swim func()
}
```

Wait a second! Where is the field name of the `Animal` type? Did you realize that I used the word *embed* in the previous paragraph? This is because, in Go, you can also embed objects within objects to make it look a lot like inheritance. That is, we won't have to explicitly call the field name to have access to its fields and method because they'll be part of us. So the following code will be perfectly okay:

```
fish := Shark{
    Swim: Swim,
}

fish.Eat()
fish.Swim()
```

Now we have an `Animal` type, which is zero-initialized and embedded. This is why I can call the `Eat` method of the `Animal` struct without creating it or using the intermediate field name. The output of this snippet is the following:

```
$ go run main.go
Eating
Swimming!
```

Finally, there is a third method to use the Composite pattern. We could create a `Swimmer` interface with a `Swim` method and a `SwimmerImpl` type to embed it in the athlete swimmer:

```
type Swimmer interface {
    Swim()
}

type Trainer interface {
    Train()
}

type SwimmerImpl struct{}
func (s *SwimmerImpl) Swim() {
    println("Swimming!")
}

type CompositeSwimmerB struct{
    Trainer
    Swimmer
}
```

With this method, you have more explicit control over object creation. The `Swimmer` field is embedded, but won't be zero-initialized as it is a pointer to an interface. The correct use of this approach will be the following:

```
swimmer := CompositeSwimmerB{
    &Athlete{},
    &SwimmerImpl{},
}

swimmer.Train()
swimmer.Swim()
```

And the output for `CompositeSwimmerB` is the following, as expected:

```
$ go run main.go
Training
Swimming!
```

Which approach is better? Well, I have a personal preference, which shouldn't be considered the rule of thumb. In my opinion, the *interfaces* approach is the best for quite a few reasons, but mainly for explicitness. First of all, you are working with interfaces which are preferred instead of structs. Second, you aren't leaving parts of your code to the zero-initialization feature of the compiler. It's a really powerful feature, but one that must be used with care, because it can lead to runtime problems which you'll find at compile time when working with interfaces. In different situations, zero-initialization will save you at runtime, in fact! But I prefer to work with interfaces as much as possible, so this is not actually one of the options.

Binary Tree compositions

Another very common approach to the Composite pattern is when working with Binary Tree structures. In a Binary Tree, you need to store instances of itself in a field:

```
type Tree struct {
    LeafValue int
    Right     *Tree
    Left      *Tree
}
```

This is some kind of recursive compositing, and, because of the nature of recursivity, we must use pointers so that the compiler knows how much memory it must reserve for this struct. Our `Tree` struct stored a `LeafValue` object for each instance and a new `Tree` in its `Right` and `Left` fields.

With this structure, we could create an object like this:

```
root := Tree{
    LeafValue: 0,
    Right:&Tree{
        LeafValue: 5,
        Right: &Tree{ 6, nil, nil },
        Left:  nil,
    },
    Left:&Tree{ 4, nil, nil },
}
```

We can print the contents of its deepest branch like this:

```
fmt.Println(root.Right.Right.LeafValue)
```

```
$ go run main.go
6
```

Composite pattern versus inheritance

When using the Composite design pattern in Go, you must be very careful not to confuse it with inheritance. For example, when you embed a `Parent` struct within a `Son` struct, like in the following example:

```
type Parent struct {
    SomeField int
}

type Son struct {
    Parent
}
```

You cannot consider that the `Son` struct is also the `Parent` struct. What this means is that you cannot pass an instance of the `Son` struct to a function that is expecting a `Parent` struct like the following:

```
func GetParentField(p *Parent) int{
    fmt.Println(p.SomeField)
}
```

When you try to pass a `Son` instance to the `GetParentField` method, you will get the following error message:

```
cannot use son (type Son) as type Parent in argument to GetParentField
```

This, in fact, makes a lot of sense. What's the solution for this? Well, you can simply composite the `Son` struct with the parent without embedding so that you can access the `Parent` instance later:

```
type Son struct {
    P Parent
}
```

So now you could use the `P` field to pass it to the `GetParentField` method:

```
son := Son{}
GetParentField(son.P)
```

Final words on the Composite pattern

At this point, you should be really comfortable using the Composite design pattern. It's a very idiomatic Go feature, and the switch from a pure object-oriented language is not very painful. The Composite design pattern makes our structures predictable but also allows us to create most of the design patterns as we will see in later chapters.

Adapter design pattern

One of the most commonly used structural patterns is the **Adapter** pattern. Like in real life, where you have plug adapters and bolt adapters, in Go, an adapter will allow us to use something that wasn't built for a specific task at the beginning.

Description

The Adapter pattern is very useful when, for example, an interface gets outdated and it's not possible to replace it easily or fast. Instead, you create a new interface to deal with the current needs of your application, which, under the hood, uses implementations of the old interface.

Adapter also helps us to maintain the *open/closed principle* in our apps, making them more predictable too. They also allow us to write code which uses some base that we can't modify.

The open/closed principle was first stated by Bertrand Meyer in his book *Object-Oriented Software Construction*. He stated that code should be open to new functionality, but closed to modifications. What does it mean? Well, it implies a few things. On one hand, we should try to write code that is extensible and not only one that works. At the same time, we should try not to modify the source code (yours or other people's) as much as we can, because we aren't always aware of the implications of this modification. Just keep in mind that extensibility in code is only possible through the use of design patterns and interface-oriented programming.



Objectives

The Adapter design pattern will help you fit the needs of two parts of the code that are incompatible at first. This is the key to being kept in mind when deciding if the Adapter pattern is a good design for your problem--two interfaces that are incompatible, but which must work together, are good candidates for an Adapter pattern (but they could also use the facade pattern, for example).

Using an incompatible interface with an Adapter object

For our example, we will have an old `Printer` interface and a new one. Users of the new interface don't expect the signature that the old one has, and we need an Adapter so that users can still use old implementations if necessary (to work with some legacy code, for example).

Requirements and acceptance criteria

Having an old interface called `LegacyPrinter` and a new one called `ModernPrinter`, create a structure that implements the `ModernPrinter` interface and can use the `LegacyPrinter` interface as described in the following steps:

1. Create an Adapter object that implements the `ModernPrinter` interface.
2. The new Adapter object must contain an instance of the `LegacyPrinter` interface.
3. When using `ModernPrinter`, it must call the `LegacyPrinter` interface under the hood, prefixing it with the text `Adapter`.

Unit testing our Printer adapter

We will write the legacy code first, but we won't test it as we should imagine that it isn't our code:

```
type LegacyPrinter interface {
    Print(s string) string
}
type MyLegacyPrinter struct { }
```

```
func (l *MyLegacyPrinter) Print(s string) (newMsg string) {
    newMsg = fmt.Sprintf("Legacy Printer: %s\n", s)
    println(newMsg)
    return
}
```

The legacy interface called `LegacyPrinter` has a `Print` method that accepts a string and returns a message. Our `MyLegacyPrinter` struct implements the `LegacyPrinter` interface and modifies the passed string by prefixing the text `Legacy Printer:`. After modifying the text, the `MyLegacyPrinter` struct prints the text on the console, and then returns it.

Now we'll declare the new interface that we'll have to adapt:

```
type ModernPrinter interface {
    PrintStored() string
}
```

In this case, the new `PrintStored` method doesn't accept any string as an argument, because it will have to be stored in the implementers in advance. We will call our Adapter pattern's `PrinterAdapter` interface:

```
type PrinterAdapter struct{
    OldPrinter LegacyPrinter
    Msg        string
}
func(p *PrinterAdapter) PrintStored() (newMsg string) {
    return
}
```

As mentioned earlier, the `PrinterAdapter` adapter must have a field to store the string to print. It must also have a field to store an instance of the `LegacyPrinter` adapter. So let's write the unit tests:

```
func TestAdapter(t *testing.T) {
    msg := "Hello World!"
```

We will use the message `Hello World!` for our adapter. When using this message with an instance of the `MyLegacyPrinter` struct, it prints the text `Legacy Printer: Hello World!:`

```
adapter := PrinterAdapter{OldPrinter: &MyLegacyPrinter{}, Msg: msg}
```

We created an instance of the `PrinterAdapter` interface called `adapter`. We passed an instance of the `MyLegacyPrinter` struct as the `LegacyPrinter` field called `OldPrinter`. Also, we set the message we want to print in the `Msg` field:

```
returnedMsg := adapter.PrintStored()

if returnedMsg != "Legacy Printer: Adapter: Hello World!\n" {
    t.Errorf("Message didn't match: %s\n", returnedMsg)
}
```

Then we used the `PrintStored` method of the `ModernPrinter` interface; this method doesn't accept any argument and must return the modified string. We know that the `MyLegacyPrinter` struct returns the passed string prefixed with the text `LegacyPrinter:`, and the adapter will prefix it with the text `Adapter:` So, in the end, we must have the text `Legacy Printer: Adapter: Hello World!\n`.

As we are storing an instance of an interface, we must also check that we handle the situation where the pointer is nil. This is done with the following test:

```
adapter = PrinterAdapter{OldPrinter: nil, Msg: msg}
returnedMsg = adapter.PrintStored()

if returnedMsg != "Hello World!" {
    t.Errorf("Message didn't match: %s\n", returnedMsg)
}
```

If we don't pass an instance of the `LegacyPrinter` interface, the Adapter must ignore its adapt nature, and simply print and return the original message. Time to run our tests; consider the following:

```
$ go test -v .
==== RUN  TestAdapter
--- FAIL: TestAdapter (0.00s)
    adapter_test.go:11: Message didn't match:
    adapter_test.go:17: Message didn't match:
FAIL
exit status 1
FAIL
```

Implementation

To make our single test pass, we must reuse the old `MyLegacyPrinter` that is stored in `PrinterAdapter` struct:

```
type PrinterAdapter struct{
    OldPrinter LegacyPrinter
    Msg        string
}

func(p *PrinterAdapter) PrintStored() (newMsg string) {
    if p.OldPrinter != nil {
        newMsg = fmt.Sprintf("Adapter: %s", p.Msg)
        newMsg = p.OldPrinter.Print(newMsg)
    } else {
        newMsg = p.Msg
    }
    return
}
```

In the `PrintStored` method, we check whether we actually have an instance of a `LegacyPrinter`. In this case, we compose a new string with the stored message and the Adapter prefix to store it in the returning variable (called `newMsg`). Then we use the pointer to the `MyLegacyPrinter` struct to print the composed message using the `LegacyPrinter` interface.

In case there is no `LegacyPrinter` instance stored in the `OldPrinter` field, we simply assign the stored message to the returning variable `newMsg` and return the method. This should be enough to pass our tests:

```
$ go test -v .
==== RUN TestAdapter
Legacy Printer: Adapter: Hello World!

--- PASS: TestAdapter (0.00s)
PASS
ok
```

Perfect! Now we can still use the old `LegacyPrinter` interface by using this Adapter while we use the `ModernPrinter` interface for future implementations. Just keep in mind that the Adapter pattern must ideally just provide the way to use the old `LegacyPrinter` and nothing else. This way, its scope will be more encapsulated and more maintainable in the future.

Examples of the Adapter pattern in Go's source code

You can find Adapter implementations at many places in the Go language's source code. The famous `http.Handler` interface has a very interesting adapter implementation. A very simple, Hello World server in Go is usually done like this:

```
package main

import (
    "fmt"
    "log"
    "net/http"
)
type MyServer struct{
    Msg string
}
func (m *MyServer) ServeHTTP(w http.ResponseWriter, r *http.Request){
    fmt.Fprintf(w, "Hello, World")
}

func main() {
    server := &MyServer{
        Msg:"Hello, World",
    }

    http.Handle("/", server)
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

The HTTP package has a function called `Handle` (like a `static` method in Java) that accepts two parameters--a string to represent the route and a `Handler` interface. The `Handler` interface is like the following:

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

We need to implement a `ServeHTTP` method that the server side of an HTTP connection will use to execute its context. But there is also a function `HandlerFunc` that allows you to define some endpoint behavior:

```
func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "Hello, World")
    })

    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

The `HandleFunc` function is actually part of an adapter for using functions directly as `ServeHTTP` implementations. Read the last sentence slowly again--can you guess how it is done?

```
type HandlerFunc func(ResponseWriter, *Request)

func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}
```

We can define a type that is a function in the same way that we define a struct. We make this function-type to implement the `ServeHTTP` method. Finally, from the `ServeHTTP` function, we call the receiver itself `f(w, r)`.

You have to think about the implicit interface implementation of Go. When we define a function like `func(ResponseWriter, *Request)`, it is implicitly being recognized as `HandlerFunc`. And because the `HandleFunc` function implements the `Handler` interface, our function implements the `Handler` interface implicitly too. Does this sound familiar to you? If $A = B$ and $B = C$, then $A = C$. Implicit implementation gives a lot of flexibility and power to Go, but you must also be careful, because you don't know if a method or function could be implementing some interface that could provoke undesirable behaviors.

We can find more examples in Go's source code. The `io` package has another powerful example with the use of pipes. A pipe in Linux is a flow mechanism that takes something on the input and outputs something else on the output. The `io` package has two interfaces, which are used everywhere in Go's source code--the `io.Reader` and the `io.Writer` interface:

```
type Reader interface {
    Read(p []byte) (n int, err error)
}

type Writer interface {
```

```
    Write(p []byte) (n int, err error)
}
```

We use `io.Reader` everywhere, for example, when you open a file using `os.OpenFile`, it returns a file, which, in fact, implements the `io.Reader` interface. Why is it useful? Imagine that you write a `Counter` struct that counts from the number you provide to zero:

```
type Counter struct {}
func (f *Counter) Count(n uint64) uint64 {
    if n == 0 {
        println(strconv.Itoa(0))
        return 0
    }

    cur := n
    println(strconv.FormatUint(cur, 10))
    return f.Count(n - 1)
}
```

If you provide the number 3 to this small snippet, it will print the following:

```
3
2
1
```

Well, not really impressive! What if I want to write to a file instead of printing? We can implement this method too. What if I want to print to a file and to the console? Well, we can implement this method too. We must modularize it a bit more by using the `io.Writer` interface:

```
type Counter struct {
    Writer io.Writer
}
func (f *Counter) Count(n uint64) uint64 {
    if n == 0 {
        f.Writer.Write([]byte(strconv.Itoa(0) + "\n"))
        return 0
    }

    cur := n
    f.Writer.Write([]byte(strconv.FormatUint(cur, 10) + "\n"))
    return f.Count(n - 1)
}
```

Now we provide an `io.Writer` in the `Writer` field. This way, we could create the counter like this: `c := Counter{os.Stdout}`, and we will get a console `Writer`. But wait a second, we haven't solved the issue where we wanted to take the count to many `Writer` consoles. But we can write a new `Adapter` with an `io.Writer` and, using a `Pipe()` to connect a reader with a writer, we can read on the opposite extreme. This way, you can solve the issue where these two interfaces, `Reader` and `Writer`, which are incompatible, can be used together.

In fact, we don't need to write the `Adapter`--the Go's `io` library has one for us in `io.Pipe()`. The pipe will allow us to convert a `Reader` to a `Writer` interface. The `io.Pipe()` method will provide us a `Writer` (the entrance of the pipe) and a `Reader` (the exit) to play with. So let's create a pipe, and assign the provided writer to the `Counter` of the preceding example:

```
pipeReader, pipeWriter := io.Pipe()
defer pw.Close()
defer pr.Close()

counter := Counter{
    Writer: pipeWriter,
}
```

Now we have a `Reader` interface where we previously had a `Writer`. Where can we use the `Reader`? The `io.TeeReader` function helps us to copy the stream of data from a `Reader` interface to the `Writer` interface and, it returns a new `Reader` that you can still use to stream data again to a second writer. So we will stream the data from the same reader to two writers--the `file` and the `Stdout`.

```
tee := io.TeeReader(pipeReader, file)
```

So now we know that we are writing to a file that we have passed to the `TeeReader` function. We still need to print to the console. The `io.Copy` adapter can be used like `TeeReader`--it takes a reader and writes its contents to a writer:

```
go func() {
    io.Copy(os.Stdout, tee)
}()
```

We have to launch the `Copy` function in a different Go routine so that the writes are performed concurrently, and one read/write doesn't block a different read/write. Let's modify the `counter` variable to make it count till 5 again:

```
counter.Count(5)
```

With this modification to the code, we get the following output:

```
$ go run counter.go
5
4
3
2
1
0
```

Okay, the count has been printed on the console. What about the file?

```
$ cat /tmp/pipe
5
4
3
2
1
0
```

Awesome! By using the `io.Pipe()` adapter provided in the Go native library, we have uncoupled our counter from its output, and we have adapted a `Writer` interface to a `Reader` one.

What the Go source code tells us about the Adapter pattern

With the Adapter design pattern, you have learned a quick way to achieve the open/close principle in your applications. Instead of modifying your old source code (something which could not be possible in some situations), you have created a way to use the old functionality with a new signature.

Bridge design pattern

The **Bridge** pattern is a design with a slightly cryptic definition from the original *Gang of Four* book. It decouples an abstraction from its implementation so that the two can vary independently. This cryptic explanation just means that you could even decouple the most basic form of functionality: decouple an object from what it does.

Description

The Bridge pattern tries to decouple things as usual with design patterns. It decouples abstraction (an object) from its implementation (the thing that the object does). This way, we can change what an object does as much as we want. It also allows us to change the abstracted object while reusing the same implementation.

Objectives

The objective of the Bridge pattern is to bring flexibility to a struct that changes often. Knowing the inputs and outputs of a method, it allows us to change code without knowing too much about it and leaving the freedom for both sides to be modified more easily.

Two printers and two ways of printing for each

For our example, we will go to a console printer abstraction to keep it simple. We will have two implementations. The first will write to the console. Having learned about the `io.Writer` interface in the previous section, we will make the second write to an `io.Writer` interface to provide more flexibility to the solution. We will also have two abstracted object users of the implementations--a `Normal` object, which will use each implementation in a straightforward manner, and a `Packt` implementation, which will append the sentence `Message from Packt:` to the printing message.

At the end of this section, we will have two abstraction objects, which have two different implementations of their functionality. So, actually, we will have 2^2 possible combinations of object functionality.

Requirements and acceptance criteria

As we mentioned previously, we will have two objects (`Packt` and `Normal` printer) and two implementations (`PrinterImpl1` and `PrinterImpl2`) that we will join by using the Bridge design pattern. More or less, we will have the following requirements and acceptance criteria:

- A `PrinterAPI` that accepts a message to print
- An implementation of the API that simply prints the message to the console
- An implementation of the API that prints to an `io.Writer` interface
- A `Printer` abstraction with a `Print` method to implement in printing types

- A normal printer object, which will implement the `Printer` and the `PrinterAPI` interface
- The normal printer will forward the message directly to the implementation
- A Packt printer, which will implement the `Printer` abstraction and the `PrinterAPI` interface
- The Packt printer will append the message `Message` from `Packt`: to all prints

Unit testing the Bridge pattern

Let's start with *acceptance criteria 1*, the `PrinterAPI` interface. Implementers of this interface must provide a `PrintMessage(string)` method that will print the message passed as an argument:

```
type PrinterAPI interface {
    PrintMessage(string) error
}
```

We will pass to *acceptance criteria 2* with an implementation of the previous API:

```
type PrinterImpl1 struct{}

func (p *PrinterImpl1) PrintMessage(msg string) error {
    return errors.New("Not implemented yet")
}
```

Our `PrinterImpl1` is a type that implements the `PrinterAPI` interface by providing an implementation of the `PrintMessage` method. The `PrintMessage` method is not implemented yet, and returns an error. This is enough to write our first unit test to cover `PrinterImpl1`:

```
func TestPrintAPI1(t *testing.T) {
    api1 := PrinterImpl1{}

    err := api1.PrintMessage("Hello")
    if err != nil {
        t.Errorf("Error trying to use the API1 implementation: Message: %s\n",
            err.Error())
    }
}
```

In our test to cover `PrintAPI1`, we created an instance of `PrinterImpl1` type. Then we used its `PrintMessage` method to print the message `Hello` to the console. As we have no implementation yet, it must return the error string `Not implemented yet`:

```
$ go test -v -run=TestPrintAPI1 .
==== RUN    TestPrintAPI1
--- FAIL: TestPrintAPI1 (0.00s)
    bridge_test.go:14: Error trying to use the API1 implementation:
Message: Not implemented yet
FAIL
exit status 1
FAIL    _C_/Users/mario/Desktop/go-design-
patterns/structural/bridge/traditional
```

Okay. Now we have to write the second API test that will work with an `io.Writer` interface:

```
type PrinterImpl2 struct{
    Writer io.Writer
}

func (d *PrinterImpl2) PrintMessage(msg string) error {
    return errors.New("Not implemented yet")
}
```

As you can see, our `PrinterImpl2` struct stores an `io.Writer` implementer. Also, our `PrintMessage` method follows the `PrinterAPI` interface.

Now that we are familiar with the `io.Writer` interface, we are going to make a test object that implements this interface, and stores whatever is written to it in a local field. This will help us check the contents that are being sent through the writer:

```
type TestWriter struct {
    Msg string
}

func (t *TestWriter) Write(p []byte) (n int, err error) {
    n = len(p)
    if n > 0 {
        t.Msg = string(p)
        return n, nil
    }
    err = errors.New("Content received on Writer was empty")
    return
}
```

In our test object, we checked that the content isn't empty before writing it to the local field. If it's empty, we return the error, and if not, we write the contents of `p` in the `Msg` field. We will use this small struct in the following tests for the second API:

```
func TestPrintAPI2(t *testing.T) {
    api2 := PrinterImpl2{}

    err := api2.PrintMessage("Hello")
    if err != nil {
        expectedErrorMessage := "You need to pass an io.Writer to PrinterImpl2"
        if !strings.Contains(err.Error(), expectedErrorMessage) {
            t.Errorf("Error message was not correct.\n"
                "Actual: %s\nExpected: %s\n", err.Error(), expectedErrorMessage)
        }
    }
}
```

Let's stop for a second here. We create an instance of `PrinterImpl2` called `api2` in the first line of the preceding code. We haven't passed any instance of `io.Writer` on purpose, so we also checked that we actually receive an error first. Then we try to use its `PrintMessage` method, but we must get an error because it doesn't have any `io.Writer` instance stored in the `Writer` field. The error must be `You need to pass an io.Writer to PrinterImpl2`, and we implicitly check the contents of the error. Let's continue with the test:

```
testWriter := TestWriter{}
api2 = PrinterImpl2{
    Writer: &testWriter,
}

expectedMessage := "Hello"
err = api2.PrintMessage(expectedMessage)
if err != nil {
    t.Errorf("Error trying to use the API2 implementation: %s\n",
        err.Error())
}

if testWriter.Msg != expectedMessage {
    t.Fatalf("API2 did not write correctly on the io.Writer. \n Actual:
%s\nExpected: %s\n", testWriter.Msg, expectedMessage)
}
```

For the second part of this unit test, we use an instance of the `TestWriter` object as an `io.Writer` interface, `testWriter`. We passed the message `Hello` to `api2`, and checked whether we receive any error. Then, we check the contents of the `testWriter.Msg` field--remember that we have written an `io.Writer` interface that stored any bytes passed to its `Write` method in the `Msg` field. If everything is correct, the message should contain the word `Hello`.

Those were our tests for `PrinterImpl2`. As we don't have any implementations yet, we should get a few errors when running this test:

```
$ go test -v -run=TestPrintAPI2 .
==== RUN TestPrintAPI2
--- FAIL: TestPrintAPI2 (0.00s)
bridge_test.go:39: Error message was not correct.
Actual: Not implemented yet
Expected: You need to pass an io.Writer to PrinterImpl2
bridge_test.go:52: Error trying to use the API2 implementation: Not
implemented yet
bridge_test.go:57: API2 did not write correctly on the io.Writer.
Actual:
Expected: Hello
FAIL
exit status 1
FAIL
```

At least one test passes--the one that checks that an error message (any) is being returned when using the `PrintMessage` without `io.Writer` being stored. Everything else fails, as expected at this stage.

Now we need a printer abstraction for objects that can use `PrinterAPI` implementers. We will define this as the `PrinterAbstraction` interface with a `Print` method. This covers the *acceptance criteria 4*:

```
type PrinterAbstraction interface {
    Print() error
}
```

For *acceptance criteria 5*, we need a normal printer. A `Printer` abstraction will need a field to store a `PrinterAPI`. So our the `NormalPrinter` could look like the following:

```
type NormalPrinter struct {
    Msg      string
    Printer  PrinterAPI
}

func (c *NormalPrinter) Print() error {
    return errors.New("Not implemented yet")
}
```

This is enough to write a unit test for the `Print()` method:

```
func TestNormalPrinter_Print(t *testing.T) {
    expectedMessage := "Hello io.Writer"

    normal := NormalPrinter{
        Msg:expectedMessage,
        Printer: &PrinterImpl1{},
    }

    err := normal.Print()
    if err != nil {
        t.Errorf(err.Error())
    }
}
```

The first part of the test checks that the `Print()` method isn't implemented yet when using `PrinterImpl1` `PrinterAPI` interface. The message we'll use along this test is `Hello io.Writer`. With the `PrinterImpl1`, we don't have an easy way to check the contents of the message, as we print directly to the console. Checking, in this case, is visual, so we can check *acceptance criteria 6*:

```
testWriter := TestWriter{}
normal = NormalPrinter{
    Msg: expectedMessage,
    Printer: &PrinterImpl2{
        Writer:&testWriter,
    },
}

err = normal.Print()
if err != nil {
    t.Error(err.Error())
}
```

```
    if testWriter.Msg != expectedMessage {
        t.Errorf("The expected message on the io.Writer doesn't match actual.\n"
        "Actual: %s\nExpected: %s\n", testWriter.Msg, expectedMessage)
    }
}
```

The second part of NormalPrinter tests uses PrinterImpl2, the one that needs an io.Writer interface implementer. We reuse our TestWriter struct here to check the contents of the message. So, in short, we want a NormalPrinter struct that accepts a Msg of type string and a Printer of type PrinterAPI. At this point, if I use the Print method, I shouldn't get any error, and the Msg field on TestWriter must contain the message we passed to NormalPrinter on its initialization.

Let's run the tests:

```
$ go test -v -run=TestNormalPrinter_Print .
==== RUN  TestNormalPrinter_Print
--- FAIL: TestNormalPrinter_Print (0.00s)
    bridge_test.go:72: Not implemented yet
    bridge_test.go:85: Not implemented yet
    bridge_test.go:89: The expected message on the io.Writer doesn't match
actual.

    Actual:
        Expected: Hello io.Writer
FAIL
exit status 1
FAIL
```

There is a trick to quickly check the validity of a unit test--the number of times we called t.Error or t.Errorf must match the number of messages of error on the console and the lines where they were produced. In the preceding test results, there are three errors at *lines* 72, 85, and 89, which exactly match the checks we wrote.

Our PacktPrinter struct will have a very similar definition to NormalPrinter at this point:

```
type PacktPrinter struct {
    Msg      string
    Printer PrinterAPI
}

func (c *PacktPrinter) Print() error {
    return errors.New("Not implemented yet")
}
```

This covers *acceptance criteria 7*. And we can almost copy and paste the contents of the previous test with a few changes:

```
func TestPacktPrinter_Print(t *testing.T) {
    passedMessage := "Hello io.Writer"
    expectedMessage := "Message from Packt: Hello io.Writer"

    packt := PacktPrinter{
        Msg:passedMessage,
        Printer: &PrinterImpl1{},
    }

    err := packt.Print()
    if err != nil {
        t.Errorf(err.Error())
    }

    testWriter := TestWriter{}
    packt = PacktPrinter{
        Msg: passedMessage,
        Printer:&PrinterImpl2{
            Writer:&testWriter,
        },
    }

    err = packt.Print()
    if err != nil {
        t.Error(err.Error())
    }

    if testWriter.Msg != expectedMessage {
        t.Errorf("The expected message on the io.Writer doesn't match actual.\nActual: %s\nExpected: %s\n", testWriter.Msg,expectedMessage)
    }
}
```

What have we changed here? Now we have `passedMessage`, which represents the message we are passing to `PacktPrinter`. We also have an `expected message` that contains the prefixed message from `Packt`. If you remember *acceptance criteria 8*, this abstraction must prefix the text `Message from Packt:` to any message that is passed to it, and, at the same time, it must be able to use any implementation of a `PrinterAPI` interface.

The second change is that we actually create `PacktPrinter` structs instead of the `NormalPrinter` structs; everything else is the same:

```
$ go test -v -run=TestPacktPrinter_Print .
==== RUN    TestPacktPrinter_Print
--- FAIL: TestPacktPrinter_Print (0.00s)
    bridge_test.go:104: Not implemented yet
    bridge_test.go:117: Not implemented yet
    bridge_test.go:121: The expected message on the io.Writer doesn't match
actual.
        Actual:
        Expected: Message from Packt: Hello io.Writer
FAIL
exit status 1
FAIL
```

Three checks, three errors. All tests have been covered, and we can finally move on to the implementation.

Implementation

We will start implementing in the same order that we created our tests, first with the `PrinterImpl1` definition:

```
type PrinterImpl1 struct{}
func (d *PrinterImpl1) PrintMessage(msg string) error {
    fmt.Printf("%s\n", msg)
    return nil
}
```

Our first API takes the message `msg` and prints it to the console. In the case of an empty string, nothing will be printed. This is enough to pass the first test:

```
$ go test -v -run=TestPrintAPI1 .
==== RUN    TestPrintAPI1
Hello
--- PASS: TestPrintAPI1 (0.00s)
PASS
ok
```

You can see the `Hello` message in the second line of the output of the test, just after the `RUN` message.

The `PrinterImpl2` struct isn't very complex either. The difference is that instead of printing to the console, we are going to write on an `io.Writer` interface, which must be stored in the struct:

```
type PrinterImpl2 struct {
    Writer io.Writer
}

func (d *PrinterImpl2) PrintMessage(msg string) error {
    if d.Writer == nil {
        return errors.New("You need to pass an io.Writer to PrinterImpl2")
    }

    fmt.Fprintf(d.Writer, "%s", msg)
    return nil
}
```

As defined in our tests, we checked the contents of the `Writer` field first and returned the expected error message `You need to pass an io.Writer to PrinterImpl2`, if nothing is stored. This is the message we'll check later in the test. Then, the `fmt.Fprintf` method takes an `io.Writer` interface as the first field and a message formatted as the rest, so we simply forward the contents of the `msg` argument to the `io.Writer` provided:

```
$ go test -v -run=TestPrintAPI2 .
==== RUN TestPrintAPI2
--- PASS: TestPrintAPI2 (0.00s)
PASS
ok
```

Now we'll continue with the normal printer. This printer must simply forward the message to the `PrinterAPI` interface stored without any modification. In our test, we are using two implementations of `PrinterAPI`--one that prints to the console and one that writes to an `io.Writer` interface:

```
type NormalPrinter struct {
    Msg      string
    Printer PrinterAPI
}

func (c *NormalPrinter) Print() error {
    c.Printer.PrintMessage(c.Msg)
    return nil
}
```

We returned nil as no error has occurred. This should be enough to pass the unit tests:

```
$ go test -v -run=TestNormalPrinter_Print .
==== RUN TestNormalPrinter_Print
Hello io.Writer
--- PASS: TestNormalPrinter_Print (0.00s)
PASS
ok
```

In the preceding output, you can see the `Hello io.Writer` message that the `PrinterImpl1` struct writes to `stdout`. We can consider this check as having passed:

Finally, the `PacktPrinter` method is similar to `NormalPrinter`, but just prefixes every message with the text `Message from Packt:`:

```
type PacktPrinter struct {
    Msg      string
    Printer  PrinterAPI
}

func (c *PacktPrinter) Print() error {
    c.Printer.PrintMessage(fmt.Sprintf("Message from Packt: %s", c.Msg))
    return nil
}
```

Like in the `NormalPrinter` method, we accepted a `Msg` string and a `PrinterAPI` implementation in the `Printer` field. Then we used the `fmt.Sprintf` method to compose a new string with the text `Message from Packt:` and the provided message. We took the composed text and passed it to the `PrintMessage` method of `PrinterAPI` stored in the `Printer` field of the `PacktPrinter` struct:

```
$ go test -v -run=TestPacktPrinter_Print .
==== RUN TestPacktPrinter_Print
Message from Packt: Hello io.Writer
--- PASS: TestPacktPrinter_Print (0.00s)
PASS
ok
```

Again, you can see the results of using `PrinterImpl1` for writing to `stdout` with the text `Message from Packt: Hello io.Writer`. This last test should cover all of our code in the Bridge pattern. As you have seen previously, you can check the coverage by using the `-cover` flag:

```
$ go test -cover .
ok
2.622s coverage: 100.0% of statements
```

Wow! 100% coverage--this looks good. However, this doesn't mean that the code is perfect. We haven't checked that the contents of the messages weren't empty, maybe something that should be avoided, but it isn't a part of our requirements, which is also an important point. Just because some feature isn't in the requirements or the acceptance criteria doesn't mean that it shouldn't be covered.

Reuse everything with the Bridge pattern

With the Bridge pattern, we have learned how to uncouple an object and its implementation for the `PrintMessage` method. This way, we can reuse its abstractions as well as its implementations. We can swap the printer abstractions as well as the printer APIs as much as we want without affecting the user code.

We have also tried to keep things as simple as possible, but I'm sure that you have realized that all implementations of the `PrinterAPI` interface could have been created using a factory. This would be very natural, and you could find many implementations that have followed this approach. However, we shouldn't get into over-engineering, but should analyze each problem to make a precise design of its needs and finds the best way to create a reusable, maintainable, and *readable* source code. Readable code is commonly forgotten, but a robust and uncoupled source code is useless if nobody can understand it to maintain it. It's like a book of the tenth century--it could be a precious story but pretty frustrating if we have difficulty understanding its grammar.

Summary

We have seen the power of composition in this chapter and many of the ways that Go takes advantage of it by its own nature. We have seen that the Adapter pattern can help us make two incompatible interfaces work together by using an `Adapter` object in between. At the same time, we have seen some real-life examples in Go's source code, where the creators of the language used this design pattern to improve the possibilities of some particular piece of the standard library. Finally, we have seen the Bridge pattern and its possibilities, allowing us to create swapping structures with complete reusability between objects and their implementations.

Also, we have used the Composite design pattern throughout the chapter, not only when explaining it. We have mentioned it earlier but design patterns make use of each other very frequently. We have used pure composition instead of embedding to increase readability, but, as you have learned, you can use both interchangeably according to your needs. We will keep using the Composite pattern in the following chapters, as it is the foundation for building relationships in the Go programming language.

4

Structural Patterns - Proxy, Facade, Decorator, and Flyweight Design Patterns

With this chapter, we will finish with the Structural patterns. We have left some of the most complex ones till the end so that you get more used to the mechanics of design patterns, and the features of Go language.

In this chapter, we will work at writing a cache to access a database, a library to gather weather data, a server with runtime middleware, and discuss a way to save memory by saving shareable states between the types values.

Proxy design pattern

We'll start the final chapter on structural patterns with the Proxy pattern. It's a simple pattern that provides interesting features and possibilities with very little effort.

Description

The Proxy pattern usually wraps an object to hide some of its characteristics. These characteristics could be the fact that it is a remote object (remote proxy), a very heavy object such as a very big image or the dump of a terabyte database (virtual proxy), or a restricted access object (protection proxy).

Objectives

The possibilities of the Proxy pattern are many, but in general, they all try to provide the same following functionalities:

- Hide an object behind the proxy so the features can be hidden, restricted, and so on
- Provide a new abstraction layer that is easy to work with, and can be changed easily

Example

For our example, we are going to create a remote proxy, which is going to be a cache of objects before accessing a database. Let's imagine that we have a database with many users, but instead of accessing the database each time we want information about a user, we will have a **First In First Out (FIFO)** stack of users in a Proxy pattern (FIFO is a way of saying that when the cache needs to be emptied, it will delete the first object that entered first).

Acceptance criteria

We will wrap an imaginary database, represented by a slice, with our Proxy pattern. Then, the pattern will have to stick to the following acceptance criteria:

1. All access to the database of users will be done through the Proxy type.
2. A stack of n number of recent users will be kept in the Proxy.
3. If a user already exists in the stack, it won't query the database, and will return the stored one
4. If the queried user doesn't exist in the stack, it will query the database, remove the oldest user in the stack if it's full, store the new one, and return it.

Unit test

Since version 1.7 of Go, we can embed tests within tests by using closures so we can group them in a more human-readable way, and reduce the number of `Test_` functions. Refer to [Chapter 1, Ready... Steady... Go!](#) to learn how to install the new version of Go if your current version is older than version 1.7.

The types for this pattern will be the proxy user and user list structs as well as a `UserFinder` interface that the database and the Proxy will implement. This is key because the Proxy must implement the same interfaces as the features of the type it tries to wrap:

```
type UserFinder interface {
    FindUser(id int32) (User, error)
}
```

The `UserFinder` is the interface that the database and the Proxy implement. The `User` is a type with a member called `ID`, which is `int32` type:

```
type User struct {
    ID int32
}
```

Finally, the `UserList` is a type of a slice of users. Consider the following syntax for that:

```
type UserList []User
```

If you are asking why we aren't using a slice of users directly, the answer is that by declaring a sequence of users this way, we can implement the `UserFinder` interface but with a slice, we can't.

Finally, the Proxy type, called `UserListProxy` will be composed of a `UserList` slice, which will be our database representation. The `StackCache` members which will also be of `UserList` type for simplicity, `StackCapacity` to give our stack the size we want.

We will cheat a bit for the purpose of this tutorial and declare a Boolean state on a field called `DidDidLastSearchUsedCache` that will hold if the last performed search has used the cache, or has accessed the database:

```
type UserListProxy struct {
    SomeDatabase UserList
    StackCache UserList
    StackCapacity int
    DidDidLastSearchUsedCache bool
}

func (u *UserListProxy) FindUser(id int32) (User, error) {
    return User{}, errors.New("Not implemented yet")
}
```

The `UserListProxy` type will cache a maximum of `StackCapacity` users, and rotate the cache if it reaches this limit. The `StackCache` members will be populated from objects from `SomeDatabase` type.

The first test is called `TestUserListProxy`, and is listed next:

```
import (
    "math/rand"
    "testing"
)

func Test_UserListProxy(t *testing.T) {
    someDatabase := UserList{}

    rand.Seed(2342342)
    for i := 0; i < 1000000; i++ {
        n := rand.Int31()
        someDatabase = append(someDatabase, User{ID: n})
    }
}
```

The preceding test creates a user list of 1 million users with random names. To do so, we feed the random number generator by calling the `Seed()` function with some constant seed so our randomized results are also constant; and the user IDs are generated from it. It might have some duplicates, but it serves our purpose.

Next, we need a proxy with a reference to `someDatabase`, which we have just created:

```
proxy := UserListProxy{
    SomeDatabase: &someDatabase,
    StackCapacity: 2,
    StackCache: UserList{},
}
```

At this point, we have a proxy object composed of a mock database with 1 million users, and a cache implemented as a FIFO stack with a size of 2. Now we will get three random IDs from `someDatabase` to use in our stack:

```
knownIDs := [3]int32 {someDatabase[3].ID,
    someDatabase[4].ID, someDatabase[5].ID}
```

We took the fourth, fifth, and sixth IDs from the slice (remember that arrays and slices start with 0, so the index 3 is actually the fourth position in the slice).

This is going to be our starting point before launching the embedded tests. To create an embedded test, we have to call the `Run` method of the `testing.T` pointer, with a description and a closure with the `func(t *testing.T)` signature:

```
t.Run("FindUser - Empty cache", func(t *testing.T) {
    user, err := proxy.FindUser(knownIDs[0])
    if err != nil {
        t.Fatal(err)
    }
})
```

```
}
```

For example, in the preceding code snippet, we give the description `FindUser - Empty cache`. Then we define our closure. First it tries to find a user with a known ID, and checks for errors. As the description implies, the cache is empty at this point, and the user will have to be retrieved from the `someDatabase` array:

```
if user.ID != knownIDs[0] {
    t.Error("Returned user name doesn't match with expected")
}

if len(proxy.StackCache) != 1 {
    t.Error("After one successful search in an empty cache, the size of it
must be one")
}

if proxy.DidLastSearchUsedCache {
    t.Error("No user can be returned from an empty cache")
}
}
```

Finally, we check whether the returned user has the same ID as that of the expected user at index 0 of the `knownIDs` slice, and that the proxy cache now has a size of 1. The state of the member `DidLastSearchUsedCache` proxy must not be `true`, or we will not pass the test. Remember, this member tells us whether the last search has been retrieved from the slice that represents a database, or from the cache.

The second embedded test for the Proxy pattern is to ask for the same user as before, which must now be returned from the cache. It's very similar to the previous test, but now we have to check if the user is returned from the cache:

```
t.Run("FindUser - One user, ask for the same user", func(t *testing.T) {
    user, err := proxy.FindUser(knownIDs[0])
    if err != nil {
        t.Fatal(err)
    }

    if user.ID != knownIDs[0] {
        t.Error("Returned user name doesn't match with expected")
    }

    if len(proxy.StackCache) != 1 {
        t.Error("Cache must not grow if we asked for an object that is stored
on it")
    }

    if !proxy.DidLastSearchUsedCache {
```

```
        t.Error("The user should have been returned from the cache")
    }
})
```

So, again we ask for the first known ID. The proxy cache must maintain a size of 1 after this search, and the `DidLastSearchUsedCache` member must be true this time, or the test will fail.

The last test will overflow the `StackCache` array on the `proxy` type. We will search for two new users that our `proxy` type will have to retrieve from the database. Our stack has a size of 2, so it will have to remove the first user to allocate space for the second and third users:

```
user1, err := proxy.FindUser(knownIDs[0])
if err != nil {
    t.Fatal(err)
}

user2, _ := proxy.FindUser(knownIDs[1])
if proxy.DidLastSearchUsedCache {
    t.Error("The user wasn't stored on the proxy cache yet")
}

user3, _ := proxy.FindUser(knownIDs[2])
if proxy.DidLastSearchUsedCache {
    t.Error("The user wasn't stored on the proxy cache yet")
}
```

We have retrieved the first three users. We aren't checking for errors because that was the purpose of the previous tests. This is important to recall that there is no need to over-test your code. If there is any error here, it will arise in the previous tests. Also, we have checked that the `user2` and `user3` queries do not use the cache; they shouldn't be stored there yet.

Now we are going to look for the `user1` query in the `Proxy`. It shouldn't exist, as the stack has a size of 2, and `user1` was the first to enter, hence, the first to go out:

```
for i := 0; i < len(proxy.StackCache); i++ {
    if proxy.StackCache[i].ID == user1.ID {
        t.Error("User that should be gone was found")
    }
}

if len(proxy.StackCache) != 2 {
    t.Error("After inserting 3 users the cache should not grow" +
    " more than to two")
}
```

It doesn't matter if we ask for a thousand users; our cache can't be bigger than our configured size.

Finally, we are going to again range over the users stored in the cache, and compare them with the last two we queried. This way, we will check that just those users are stored in the cache. Both must be found on it:

```
for _, v := range proxy.StackCache {
    if v != user2 && v != user3 {
        t.Error("A non expected user was found on the cache")
    }
}
```

Running the tests now should give some errors, as usual. Let's run them now:

```
$ go test -v .
=== RUN   Test_UserListProxy
=== RUN   Test_UserListProxy/FindUser--Empty_cache
=== RUN   Test_UserListProxy/FindUser--One_user,_ask_for_the_same_user
--- FAIL: Test_UserListProxy/FindUser--overflowing_the_stack (0.06s)
    --- FAIL: Test_UserListProxy/FindUser--Empty_cache (0.00s)
        proxy_test.go:28: Not implemented yet
    --- FAIL: Test_UserListProxy/FindUser--
        _One_user,_ask_for_the_same_user (0.00s)
            proxy_test.go:47: Not implemented yet
    --- FAIL: Test_UserListProxy/FindUser--overflowing_the_stack (0.00s)
        proxy_test.go:66: Not implemented yet
FAIL
exit status 1
FAIL
```

So, let's implement the `FindUser` method to act as our Proxy.

Implementation

In our Proxy, the `FindUser` method will search for a specified ID in the cache list. If it finds it, it will return the ID. If not, it will search in the database. Finally, if it's not in the database list, it will return an error.

If you remember, our Proxy pattern is composed of two `UserList` types (one of them a pointer), which are actually slices of `User` type. We will implement a `FindUser` method in `User` type too, which, by the way, has the same signature as the `UserFinder` interface:

```
type UserList []User

func (t *UserList) FindUser(id int32) (User, error) {
    for i := 0; i < len(*t); i++ {
        if (*t)[i].ID == id {
            return (*t)[i], nil
        }
    }
    return User{}, fmt.Errorf("User %s could not be found\n", id)
}
```

The `FindUser` method in the `UserList` slice will iterate over the list to try and find a user with the same ID as the `id` argument, or return an error if it can't find it.

You may be wondering why the pointer `t` is between parentheses. This is to dereference the underlying array before accessing its indexes. Without it, you'll have a compilation error, because the compiler tries to search the index before dereferencing the pointer.

So, the first part of the proxy `FindUser` method can be written as follows:

```
func (u *UserListProxy) FindUser(id int32) (User, error) {
    user, err := u.StackCache.FindUser(id)
    if err == nil {
        fmt.Println("Returning user from cache")
        u.DidLastSearchUsedCache = true
        return user, nil
    }
}
```

We use the preceding method to search for a user in the `StackCache` member. The error will be nil if it can find it, so we check this to print a message to the console, change the state of `DidLastSearchUsedCache` to `true` so that the test can check whether the user was retrieved from cache, and finally, return the user.

So, if the error was not nil, it means that it couldn't find the user in the stack. So, the next step is to search in the database:

```
user, err = u.SomeDatabase.FindUser(id)
if err != nil {
    return User{}, err
}
```

We can reuse the `FindUser` method we wrote for `UserList` database in this case, because both have the same type for the purpose of this example. Again, it searches the user in the database represented by the `UserList` slice, but in this case, if the user isn't found, it returns the error generated in `UserList`.

When the user is found (`err` is `nil`), we have to add the user to the stack. For this purpose, we write a dedicated private method that receives a pointer of type `UserListProxy`:

```
func (u *UserListProxy) addUserToStack(user User) {
    if len(u.StackCache) >= u.StackCapacity {
        u.StackCache = append(u.StackCache[1:], user)
    } else {
        u.StackCache addUser(user)
    }
}

func (t *UserList) addUser(newUser User) {
    *t = append(*t, newUser)
}
```

The `addUserToStack` method takes the `user` argument, and adds it to the stack in place. If the stack is full, it removes the first element in it before adding. We have also written an `addUser` method to `UserList` to help us in this. So, now in `FindUser` method, we just have to add one line:

```
u.addUserToStack(user)
```

This adds the new user to the stack, removing the last if necessary.

Finally, we just have to return the new user of the stack, and set the appropriate value on `DidLastSearchUsedCache` variable. We also write a message to the console to help in the testing process:

```
fmt.Println("Returning user from database")
u.DidLastSearchUsedCache = false
return user, nil
}
```

With this, we have enough to pass our tests:

```
$ go test -v .
== RUN Test_UserListProxy
== RUN Test_UserListProxy/FindUser_-_Empty_cache
Returning user from database
== RUN Test_UserListProxy/FindUser_-_One_user,_ask_for_the_same_user
Returning user from cache
```

```
== RUN Test_UserListProxy/FindUser--overflowing_the_stack
Returning user from cache
Returning user from database
Returning user from database
--- PASS: Test_UserListProxy (0.09s)
--- PASS: Test_UserListProxy/FindUser--Empty_cache (0.00s)
--- PASS: Test_UserListProxy/FindUser--One_user,_ask_for_the_same_user
(0.00s)
--- PASS: Test_UserListProxy/FindUser--overflowing_the_stack (0.00s)
PASS
ok
```

You can see in the preceding messages that our Proxy has worked flawlessly. It has returned the first search from the database. Then, when we search for the same user again, it uses the cache. Finally, we made a new test that calls three different users and we can observe, by looking at the console output, that just the first was returned from the cache and that the other two were fetched from the database.

Proxying around actions

Wrap proxies around types that need some intermediate action, like giving authorization to the user or providing access to a database, like in our example.

Our example is a good way to separate application needs from database needs. If our application accesses the database too much, a solution for this is not in your database. Remember that the Proxy uses the same interface as the type it wraps, and, for the user, there shouldn't be any difference between the two.

Decorator design pattern

We'll continue this chapter with the big brother of the Proxy pattern, and maybe, one of the most powerful design patterns of all. The **Decorator** pattern is pretty simple, but, for instance, it provides a lot of benefits when working with legacy code.

Description

The Decorator design pattern allows you to decorate an already existing type with more functional features without actually touching it. How is it possible? Well, it uses an approach similar to *matryoshka dolls*, where you have a small doll that you can put inside a doll of the same shape but bigger, and so on and so forth.

The Decorator type implements the same interface of the type it decorates, and stores an instance of that type in its members. This way, you can stack as many decorators (dolls) as you want by simply storing the old decorator in a field of the new one.

Objectives

When you think about extending legacy code without the risk of breaking something, you should think of the Decorator pattern first. It's a really powerful approach to deal with this particular problem.

A different field where the Decorator is very powerful may not be so obvious though it reveals itself when creating types with lots of features based on user inputs, preferences, or similar inputs. Like in a Swiss knife, you have a base type (the frame of the knife), and from there you unfold its functionalities.

So, precisely when are we going to use the Decorator pattern? Answer to this question:

- When you need to add functionality to some code that you don't have access to, or you don't want to modify to avoid a negative effect on the code, and follow the open/close principle (like legacy code)
- When you want the functionality of an object to be created or altered dynamically, and the number of features is unknown and could grow fast

Example

In our example, we will prepare a `Pizza` type, where the core is the pizza and the ingredients are the decorating types. We will have a couple of ingredients for our pizza-- onion and meat.

Acceptance criteria

The acceptance criteria for a Decorator pattern is to have a common interface and a core type, the one that all layers will be built over:

- We must have the main interface that all decorators will implement. This interface will be called `IngredientAdd`, and it will have the `AddIngredient()` string method.
- We must have a core `PizzaDecorator` type (the decorator) that we will add ingredients to.

- We must have an ingredient "onion" implementing the same `IngredientAdd` interface that will add the string `onion` to the returned pizza.
- We must have a ingredient "meat" implementing the `IngredientAdd` interface that will add the string `meat` to the returned pizza.
- When calling `AddIngredient` method on the top object, it must return a fully decorated pizza with the text `Pizza` with the following ingredients: `meat, onion`.

Unit test

To launch our unit tests, we must first create the basic structures described in accordance with the acceptance criteria. To begin with, the interface that all decorating types must implement is as follows:

```
type IngredientAdd interface {
    AddIngredient() (string, error)
}
```

The following code defines the `PizzaDecorator` type, which must have `IngredientAdd` inside, and which implements `IngredientAdd` too:

```
type PizzaDecorator struct{
    Ingredient IngredientAdd
}

func (p *PizzaDecorator) AddIngredient() (string, error) {
    return "", errors.New("Not implemented yet")
}
```

The definition of the `Meat` type will be very similar to that of the `PizzaDecorator` structure:

```
type Meat struct {
    Ingredient IngredientAdd
}

func (m *Meat) AddIngredient() (string, error) {
    return "", errors.New("Not implemented yet")
}
```

Now we define the `Onion` struct in a similar fashion:

```
type Onion struct {
    Ingredient IngredientAdd
}
```

```
}

func (o *Onion) AddIngredient() (string, error) {
    return "", errors.New("Not implemented yet")
}
```

This is enough to implement the first unit test, and to allow the compiler to run them without any compiling errors:

```
func TestPizzaDecorator_AddIngredient(t *testing.T) {
    pizza := &PizzaDecorator{}
    pizzaResult, _ := pizza.AddIngredient()
    expectedText := "Pizza with the following ingredients:"
    if !strings.Contains(pizzaResult, expectedText) {
        t.Errorf("When calling the add ingredient of the pizza decorator it
must return the text %s the expected text, not '%s'", pizzaResult,
expectedText)
    }
}
```

Now it must compile without problems, so we can check that the test fails:

```
$ go test -v -run=TestPizzaDecorator .
==== RUN  TestPizzaDecorator_AddIngredient
--- FAIL: TestPizzaDecorator_AddIngredient (0.00s)
decorator_test.go:29: Not implemented yet
decorator_test.go:34: When the the AddIngredient method of the pizza
decorator object is called, it must return the text Pizza with the
following ingredients:
FAIL
exit status 1
FAIL
```

Our first test is done, and we can see that the `PizzaDecorator` struct isn't returning anything yet, that's why it fails. We can now move on to the `Onion` type. The test of the `Onion` type is quite similar to that of the `Pizza` decorator, but we must also make sure that we actually add the ingredient to the `IngredientAdd` method and not to a nil pointer:

```
func TestOnion_AddIngredient(t *testing.T) {
    onion := &Onion{}
    onionResult, err := onion.AddIngredient()
    if err == nil {
        t.Errorf("When calling AddIngredient on the onion decorator without" +
"an IngredientAdd on its Ingredient field must return an error, not a
string with '%s'", onionResult)
    }
}
```

The first half of the preceding test examines the returning error when no `IngredientAdd` method is passed to the `Onion` struct initializer. As no pizza is available to add the ingredient, an error must be returned:

```
onion = &Onion{&PizzaDecorator{}}
onionResult, err = onion.AddIngredient()

if err != nil {
    t.Error(err)
}
if !strings.Contains(onionResult, "onion") {
    t.Errorf("When calling the add ingredient of the onion decorator it" +
    "must return a text with the word 'onion', not '%s'", onionResult)
}
```

The second part of the `Onion` type test actually passes `PizzaDecorator` structure to the initializer. Then, we check whether no error is being returned, and also whether the returning string contains the word `onion` in it. This way, we can ensure that onion has been added to the pizza.

Finally for the `Onion` type, the console output of this test with our current implementation will be the following:

```
$ go test -v -run=TestOnion_AddIngredient .
==== RUN TestOnion_AddIngredient
--- FAIL: TestOnion_AddIngredient (0.00s)
decorator_test.go:48: Not implemented yet
decorator_test.go:52: When calling the add ingredient of the onion
decorator it must return a text with the word 'onion', not ''
FAIL
exit status 1
FAIL
```

The `meat` ingredient is exactly the same, but we change the type to meat instead of onion:

```
func TestMeat_AddIngredient(t *testing.T) {
    meat := &Meat{}
    meatResult, err := meat.AddIngredient()
    if err == nil {
        t.Errorf("When calling AddIngredient on the meat decorator without" +
        "an IngredientAdd in its Ingredient field must return an error," + "not a
        string with '%s'", meatResult)
    }

    meat = &Meat{&PizzaDecorator{}}
    meatResult, err = meat.AddIngredient()
```

```
if err != nil {
    t.Error(err)
}

if !strings.Contains(meatResult, "meat") {
    t.Errorf("When calling the add ingredient of the meat decorator it" +
"must return a text with the word 'meat', not '%s'", meatResult)
}
}
```

So, the result of the tests will be similar:

```
go test -v -run=TestMeat_AddIngredient .
==== RUN    TestMeat_AddIngredient
---- FAIL: TestMeat_AddIngredient (0.00s)
decorator_test.go:68: Not implemented yet
decorator_test.go:72: When calling the add ingredient of the meat
decorator it must return a text with the word 'meat', not ''
FAIL
exit status 1
FAIL
```

Finally, we must check the full stack test. Creating a pizza with onion and meat must return the text Pizza with the following ingredients: meat, onion:

```
func TestPizzaDecorator_FullStack(t *testing.T) {
    pizza := &Onion{&Meat{&PizzaDecorator{}}}
    pizzaResult, err := pizza.AddIngredient()
    if err != nil {
        t.Error(err)
    }

    expectedText := "Pizza with the following ingredients: meat, onion"
    if !strings.Contains(pizzaResult, expectedText){
        t.Errorf("When asking for a pizza with onion and meat the returned " +
"string must contain the text '%s' but '%s' didn't have it",
expectedText,pizzaResult)
    }

    t.Log(pizzaResult)
}
```

Our test creates a variable called `pizza` which, like the *matryoshka dolls*, embeds types of the `IngredientAdd` method in several levels. Calling the `AddIngredient` method executes the method at the "onion" level, which executes the "meat" one, which, finally, executes that of the `PizzaDecorator` struct. After checking that no error had been returned, we check whether the returned text follows the needs of the *acceptance criteria 5*. The tests are run with the following command:

```
go test -v -run=TestPizzaDecorator_FullStack .
==== RUN TestPizzaDecorator_FullStack
==== FAIL: TestPizzaDecorator_FullStack (0.
decorator_test.go:80: Not implemented yet
decorator_test.go:87: When asking for a pizza with onion and meat the
returned string must contain the text 'Pizza with the following
ingredients: meat, onion' but '' didn't have it
FAIL
exit status 1
FAIL
```

From the preceding output, we can see that the tests now return an empty string for our decorated type. This is, of course, because no implementation has been done yet. This was the last test to check the fully decorated implementation. Let's look closely at the implementation then.

Implementation

We are going to start implementing the `PizzaDecorator` type. Its role is to provide the initial text of the full pizza:

```
type PizzaDecorator struct {
    Ingredient IngredientAdd
}

func (p *PizzaDecorator) AddIngredient() (string, error) {
    return "Pizza with the following ingredients:", nil
}
```

A single line change on the return of the `AddIngredient` method was enough to pass the test:

```
go test -v -run=TestPizzaDecorator_Add .
==== RUN TestPizzaDecorator_AddIngredient
==== PASS: TestPizzaDecorator_AddIngredient (0.00s)
PASS
ok
```

Moving on to the `Onion` struct implementation, we must take the beginning of our `IngredientAdd` returned string, and add the word `onion` at the end of it in order to get a composed pizza in return:

```
type Onion struct {
    Ingredient IngredientAdd
}

func (o *Onion) AddIngredient() (string, error) {
    if o.Ingredient == nil {
        return "", errors.New("An IngredientAdd is needed in the Ingredient field of the Onion")
    }
    s, err := o.Ingredient.AddIngredient()
    if err != nil {
        return "", err
    }
    return fmt.Sprintf("%s %s,", s, "onion"), nil
}
```

Checking that we actually have a pointer to `IngredientAdd` first, we use the contents of the inner `IngredientAdd`, and check it for errors. If no errors occur, we receive a new string composed of this content, a space, and the word `onion` (and no errors). Looks good enough to run the tests:

```
go test -v -run=TestOnion_AddIngredient .
==== RUN TestOnion_AddIngredient
--- PASS: TestOnion_AddIngredient (0.00s)
PASS
ok
```

Implementation of the `Meat` struct is very similar:

```
type Meat struct {
    Ingredient IngredientAdd
}

func (m *Meat) AddIngredient() (string, error) {
    if m.Ingredient == nil {
        return "", errors.New("An IngredientAdd is needed in the Ingredient field of the Meat")
    }
    s, err := m.Ingredient.AddIngredient()
    if err != nil {
        return "", err
    }
    return fmt.Sprintf("%s %s,", s, "meat"), nil
}
```

}

And here goes their test execution:

```
go test -v -run=TestMeat_AddIngredient .
==== RUN TestMeat_AddIngredient
--- PASS: TestMeat_AddIngredient (0.00s)
PASS
ok
```

Okay. So, now all the pieces are to be tested separately. If everything is okay, the test of the *full stacked* solution must be passing smoothly:

```
go test -v -run=TestPizzaDecorator_FullStack .
==== RUN TestPizzaDecorator_FullStack
--- PASS: TestPizzaDecorator_FullStack (0.00s)
decorator_test.go:92: Pizza with the following ingredients: meat,
onion,
PASS
ok
```

Awesome! With the Decorator pattern, we could keep stacking `IngredientAdd`s which call their inner pointer to add functionality to `PizzaDecorator`. We aren't touching the core type either, nor modifying or implementing new things. All the new features are implemented by an external type.

A real-life example - server middleware

By now, you should have understood how the Decorator pattern works. Now we can try a more advanced example using the small HTTP server that we designed in the Adapter pattern section. You learned that an HTTP server can be created by using the `http` package, and implementing the `http.Handler` interface. This interface has only one method called `ServeHTTP` (`http.ResponseWriter, http.Request`). Can we use the Decorator pattern to add more functionality to a server? Of course!

We will add a couple of pieces to this server. First, we are going to log every connection made to it to the `io.Writer` interface (for the sake of simplicity, we'll use the `io.Writer` implementation of the `os.Stdout` interface so that it outputs to the console). The second piece will add basic HTTP authentication to every request made to the server. If the authentication passes, a `Hello Decorator!` message will appear. Finally, the user will be able to select the number of decoration items that he/she wants in the server, and the server will be structured and created at runtime.

Starting with the common interface, `http.Handler`

We already have the common interface that we will decorate using nested types. We first need to create our core type, which is going to be the `Handler` that returns the sentence `Hello Decorator!`:

```
type MyServer struct{ }

func (m *MyServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Hello Decorator!")
}
```

This handler can be attributed to the `http.Handle` method to define our first endpoint. Let's check this now by creating the package's main function, and sending a GET request to it:

```
func main() {
    http.Handle("/", &MyServer{})

    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Execute the server using the Terminal to execute the `go run main.go` command. Then, open a new Terminal to make the GET request. We'll use the `curl` command to make our requests:

```
$ curl http://localhost:8080
Hello Decorator!
```

We have crossed the first milestone of our decorated server. The next step is to decorate it with logging capabilities. To do so, we must implement the `http.Handler` interface, in a new type, as follows:

```
type LoggerServer struct {
    Handler    http.Handler
    LogWriter io.Writer
}

func (s *LoggerServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(s.LogWriter, "Request URI: %s\n", r.RequestURI)
    fmt.Fprintf(s.LogWriter, "Host: %s\n", r.Host)
    fmt.Fprintf(s.LogWriter, "Content Length: %d\n",
        r.ContentLength)
    fmt.Fprintf(s.LogWriter, "Method: %s\n",
        r.Method)
    fmt.Fprintf(s.LogWriter, "-----\n")

    s.Handler.ServeHTTP(w, r)
}
```

```
}
```

We call this type `LoggerServer`. As you can see, it stores not only a `Handler`, but also `io.Writer` to write the output of the log. Our implementation of the `ServeHTTP` method prints the request URI, the host, the content length, and the used method `io.Writer`. Once printing is finished, it calls the `ServeHTTP` function of its inner `Handler` field.

We can decorate `MyServer` with this `LoggerMiddleware`:

```
func main() {
    http.Handle("/", &LoggerServer{
        LogWriter:os.Stdout,
        Handler:&MyServer{},
    })

    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Now run the `curl` command:

```
$ curl http://localhost:8080
Hello Decorator!
```

Our `curl` command returns the same message, but if you look at the Terminal where you have run the Go application, you can see the logging:

```
$ go run server_decorator.go
Request URI: /
Host: localhost:8080
Content Length: 0
Method: GET
```

We have decorated `MyServer` with logging capabilities without actually modifying it. Can we do the same with authentication? Of course! After logging the request, we will authenticate it by using **HTTP Basic Authentication** as follows:

```
type BasicAuthMiddleware struct {
    Handler http.Handler
    User     string
    Password string
}
```

The **BasicAuthMiddleware** middleware stores three fields--a handler to decorate like in the previous middlewares, a user, and a password, which will be the only authorization to access the contents on the server. The implementation of the `Decorating` method will proceed as follows:

```
func (s *BasicAuthMiddleware) ServeHTTP(w http.ResponseWriter, r
*http.Request) {
    user, pass, ok := r.BasicAuth()

    if ok {
        if user == s.User && pass == s.Password {
            s.Handler.ServeHTTP(w, r)
        } else {
            fmt.Fprintf(w, "User or password incorrect\n")
        }
    } else {
        fmt.Println(w, "Error trying to retrieve data from Basic auth")
    }
}
```

In the preceding implementation, we use the `BasicAuth` method from `http.Request` to automatically retrieve the user and password from the request, plus an `ok/ko` from the parsing action. Then we check whether the parsing is correct (returning a message to the requester if incorrect, and finishing the request). If no problems have been detected during parsing, we check whether the username and the password match with the ones stored in `BasicAuthMiddleware`. If the credentials are valid, we shall call the decorated type (our server), but if the credentials aren't valid, we receive the `User or password incorrect` message in return, and the request is finished.

Now, we need to provide the user with a way to choose among different types of servers. We will retrieve user input data in the main function. We'll have three options to choose from:

- Simple server
- Server with logging
- Server with logging and authentication

We have to use the `Fscanf` function to retrieve input from the user:

```
func main() {
    fmt.Println("Enter the type number of server you want to launch from the
following:")
    fmt.Println("1.- Plain server")
    fmt.Println("2.- Server with logging")
    fmt.Println("3.- Server with logging and authentication")

    var selection int
    fmt.Fscanf(os.Stdin, "%d", &selection)
```

```
}
```

The `Fscanf` function needs an `io.Reader` implementor as the first argument (which is going to be the input in the console), and it takes the server selected by the user from it. We'll pass `os.Stdin` as the `io.Reader` interface to retrieve user input. Then, we'll write the type of data it is going to parse. The `%d` specifier refers to an integer number. Finally, we'll write memory direction to store the parsed input, in this case, the memory position of the `selection` variable.

Once the user selects an option, we can take the basic server and decorate it at runtime, switching over to the selected option:

```
switch selection {
case 1:
    mySuperServer = new(MyServer)
case 2:
    mySuperServer = &LoggerMiddleware{
        Handler: new(MyServer),
        LogWriter: os.Stdout,
    }
case 3:
    var user, password string

    fmt.Println("Enter user and password separated by a space")
    fmt.Fscanf(os.Stdin, "%s %s", &user, &password)

    mySuperServer = &LoggerMiddleware{
        Handler: &SimpleAuthMiddleware{
            Handler: new(MyServer),
            User: user,
            Password: password,
        },
        LogWriter: os.Stdout,
    }
default:
    mySuperServer = new(MyServer)
}
```

The first option will be handled by the default `switch` option--a plain `MyServer`. In the case of the second option, we decorate a plain server with logging. The third Option is a bit more developed--we ask the user for a username and a password using `Fscanf` again. Note that you can scan more than one input, as we are doing to retrieve the user and the password. Then, we take the basic server, decorate it with authentication, and finally, with logging.

If you follow the indentation of the nested types of option three, the request passes through the logger, then the authentication middleware, and finally, the `MyServer` argument if everything is okay. The requests will follow the same route.

The end of the main function takes the decorated handler, and launches the server on the 8080 port:

```
http.Handle("/", mySuperServer)
log.Fatal(http.ListenAndServe(":8080", nil))
```

So, let's launch the server with the third option:

```
$go run server_decorator.go
Enter the server type number you want to launch from the following:
1.- Plain server
2.- Server with logging
3.- Server with logging and authentication

Enter user and password separated by a space
mario castro
```

We will first test the plain server by choosing the first option. Run the server with the command `go run server_decorator.go`, and select the first option. Then, in a different Terminal, run the basic request with curl, as follows:

```
$ curl http://localhost:8080
Error trying to retrieve data from Basic auth
```

Uh, oh! It doesn't give us access. We haven't passed any user and password, so it tells us that we cannot continue. Let's try with some random user and password:

```
$ curl -u no:correct http://localhost:8080
User or password incorrect
```

No access! We can also check in the Terminal where we launched the server and where every request is being logged:

```
Request URI: /
Host: localhost:8080
Content Length: 0
Method: GET
```

Finally, enter the correct username and password:

```
$ curl -u packt:publishing http://localhost:8080
Hello Decorator!
```

Here we are! Our request has also been logged, and the server has granted access to us. Now we can improve our server as much as we want by writing more middlewares to decorate the server's functionality.

A few words about Go's structural typing

Go has a feature that most people dislike at the beginning--structural typing. This is when your structure defines your type without explicitly writing it. For example, when you implement an interface, you don't have to write explicitly that you are actually implementing it, contrary to languages such as Java where you have to write the keyword `implements`. If your method follows the signature of the interface, you are actually implementing the interface. This can also lead to accidental implementations of interface, something that could provoke an impossible-to-track mistake, but that is very unlikely.

However, structural typing also allows you to define an interface after defining their implementers. Imagine a `MyPrinter` struct as follows:

```
type MyPrinter struct{}  
func(m *MyPrinter)Print(){  
    println("Hello")  
}
```

Imagine we have been working with the `MyPrinter` type for few months now, but it didn't implement any interface, so it can't be a possible candidate for a Decorator pattern, or maybe it can? What if we wrote an interface that matches its `Print` method after a few months? Consider the following code snippet:

```
type Printer interface {  
    Print()  
}
```

It actually implements the `Printer` interface, and we can use it to create a Decorator solution.

Structural typing allows a lot of flexibility when writing programs. If you don't know whether a type should be a part of an interface or not, you can leave it and add the interface later, when you are completely sure about it. This way, you can decorate types very easily and with little modification in your source code.

Summarizing the Decorator design pattern - Proxy versus Decorator

You might be wondering, what's the difference between the Decorator pattern and the Proxy pattern? In the Decorator pattern, we decorate a type dynamically. This means that the decoration may or may not be there, or it may be composed of one or many types. If you remember, the Proxy pattern wraps a type in a similar fashion, but it does so at compile time and it's more like a way to access some type.

At the same time, a decorator might implement the entire interface that the type it decorates also implements **or not**. So you can have an interface with 10 methods and a decorator that just implements one of them and it will still be valid. A call on a method not implemented by the decorator will be passed to the decorated type. This is a very powerful feature but also very prone to undesired behaviors at runtime if you forget to implement any interface method.

In this aspect, you may think that the Proxy pattern is less flexible, and it is. But the Decorator pattern is weaker, as you could have errors at runtime, which you can avoid at compile time by using the Proxy pattern. Just keep in mind that the Decorator is commonly used when you want to add functionality to an object at runtime, like in our web server. It's a compromise between what you need and what you want to sacrifice to achieve it.

Facade design pattern

The next pattern we'll see in this chapter is the Facade pattern. When we discussed the Proxy pattern, you got to know that it was a way to wrap a type to hide some of its features of complexity from the user. Imagine that we group many proxies in a single point such as a file or a library. This could be a Facade pattern.

Description

A facade, in architectural terms, is the front wall that hides the rooms and corridors of a building. It protects its inhabitants from cold and rain, and provides them privacy. It orders and divides the dwellings.

The Facade design pattern does the same, but in our code. It shields the code from unwanted access, orders some calls, and hides the complexity scope from the user.

Objectives

You use Facade when you want to hide the complexity of some tasks, especially when most of them share utilities (such as authentication in an API). A library is a form of facade, where someone has to provide some methods for a developer to do certain things in a friendly way. This way, if a developer needs to use your library, he doesn't need to know all the inner tasks to retrieve the result he/she wants.

So, you use the Facade design pattern in the following scenarios:

- When you want to decrease the complexity of some parts of our code. You hide that complexity behind the facade by providing a more easy-to-use method.
- When you want to group actions that are cross-related in a single place.
- When you want to build a library so that others can use your products without worrying about how it all works.

Example

As an example, we are going to take the first steps toward writing our own library that accesses OpenWeatherMaps service. In case you are not familiar with OpenWeatherMap service, it is an HTTP service that provides you with live information about weather, as well as historical data on it. The **HTTP REST API** is very easy to use, and will be a good example on how to create a Facade pattern for hiding the complexity of the network connections behind the REST service.

Acceptance criteria

The OpenWeatherMap API gives lots of information, so we are going to focus on getting live weather data in one city in some geo-located place by using its latitude and longitude values. The following are the requirements and acceptance criteria for this design pattern:

1. Provide a single type to access the data. All information retrieved from OpenWeatherMap service will pass through it.
2. Create a way to get the weather data for some city of some country.
3. Create a way to get the weather data for some latitude and longitude position.
4. Only second and third point must be visible outside of the package; everything else must be hidden (including all connection-related data).

Unit test

To start with our API Facade, we will need an interface with the methods asked in *acceptance criteria 2* and *acceptance criteria 3*:

```
type CurrentWeatherDataRetriever interface {
    GetByCityAndCountryCode(city, countryCode string) (Weather, error)
    GetByGeoCoordinates(lat, lon float32) (Weather, error)
}
```

We will call *acceptance criteria 2* `GetByCityAndCountryCode`; we will also need a city name and a country code in the string format. A country code is a two-character code, which represents the **International Organization for Standardization (ISO)** name of world countries. It returns a `Weather` value, which we will define later, and an error if something goes wrong.

Acceptance criteria 3 will be called `GetByGeoCoordinates`, and will need latitude and longitude values in the `float32` format. It will also return a `Weather` value and an error. The `Weather` value is going to be defined according to the returned JSON that the OpenWeatherMap API works with. You can find the description of this JSON at the webpage http://openweathermap.org/current#current_JSON.

If you look at the JSON definition, it has the following type:

```
type Weather struct {
    ID      int     `json:"id"`
    Name   string  `json:"name"`
    Cod    int     `json:"cod"`
    Coord struct {
        Lon float32 `json:"lon"`
        Lat float32 `json:"lat"`
    } `json:"coord"`
    Weather []struct {
        Id          int     `json:"id"`
        Main        string  `json:"main"`
        Description string  `json:"description"`
        Icon        string  `json:"icon"`
    } `json:"weather"`

    Base string `json:"base"`
    Main struct {
        Temp      float32 `json:"temp"`
        Pressure float32 `json:"pressure"`
        Humidity float32 `json:"humidity"`
        TempMin  float32 `json:"temp_min"`
        TempMax  float32 `json:"temp_max"`
    }
}
```

```
    } `json:"main"`

    Wind struct {
        Speed float32 `json:"speed"`
        Deg   float32 `json:"deg"`
    } `json:"wind"`

    Clouds struct {
        All int `json:"all"`
    } `json:"clouds"`
    Rain struct {
        ThreeHours float32 `json:"3h"`
    } `json:"rain"`
    Dt  uint32 `json:"dt"`
    Sys struct {
        Type   int      `json:"type"`
        ID     int      `json:"id"`
        Message float32 `json:"message"`
        Country string  `json:"country"`
        Sunrise int     `json:"sunrise"`
        Sunset  int     `json:"sunset"`
    } `json:"sys"`
}
```

It's quite a long struct, but we have everything that a response could include. The struct is called `Weather`, as it is composed of an `ID`, a name and a `Code (Cod)`, and a few anonymous structs, which are: `Coord`, `Weather`, `Base`, `Main`, `Wind`, `Clouds`, `Rain`, `Dt`, and `Sys`. We could write these anonymous structs outside of the `Weather` struct by giving them a name, but it would only be useful if we have to work with them separately.

After every member and struct within our `Weather` struct, you can find a ``json:"something"` line. This comes in handy when differentiating between the JSON key name and your member name. If the JSON key is `something`, we aren't forced to call our member `something`. For example, our `ID` member will be called `id` in the JSON response.

Why don't we give the name of the JSON keys to our types? Well, if your fields in your type are lowercase, the `encoding/json` package won't parse them correctly. Also, that last annotation provides us a certain flexibility, not only in terms of changing the members' names, but also of omitting some key if we don't need it, with the following signature:

```
`json:"something",omitempty"
```

With `omitempty` at the end, the parse won't fail if this key is not present in the bytes representation of the JSON key.

Okay, our acceptance criteria 1 ask for a single point of access to the API. This is going to be called `CurrentWeatherData`:

```
type CurrentWeatherData struct {
    APIkey string
}
```

The `CurrentWeatherData` type has an API key as public member to work. This is because you have to be a registered user in OpenWeatherMap to enjoy their services. Refer to the OpenWeatherMap API's webpage for documentation on how to get an API key. We won't need it in our example, because we aren't going to do integration tests.

We need mock data so that we can write a `mock` function to retrieve the data. When sending an HTTP request, the response is contained in a member called `body` in the form of an `io.Reader`. We have already worked with types that implement the `io.Reader` interface, so this should look familiar to you. Our `mock` function appears like this:

```
func getMockData() io.Reader {
    response := `{
        "coord":{"lon":-3.7,"lat":40.42}, "weather :
        [{"id":803,"main":"Clouds","description":"broken
        clouds","icon":"04n"}], "base":"stations", "main":{"temp":303.56,"pressure":1
        016.46,"humidity":26.8,"temp_min":300.95,"temp_max":305.93}, "wind":{"speed"
        :3.17,"deg":151.001}, "rain":{"3h":0.0075}, "clouds":{"all":68}, "dt":14712958
        23, "sys":{"type":3,"id":1442829648,"message":0.0278,"country":"ES","sunrise"
        :1471238808,"sunset":1471288232}, "id":3117735, "name":"Madrid", "cod":200}`

    r := bytes.NewReader([]byte(response))
    return r
}
```

This preceding mocked data was produced by making a request to OpenWeatherMap using an API key. The `response` variable is a string containing a JSON response. Take a close look at the grave accent (`) used to open and close the string. This way, you can use as many quotes as you want without any problem.

Further on, we use a special function in the `bytes` package called `NewReader`, which accepts an slice of bytes (which we create by converting the type from `string`), and returns an `io.Reader` implementor with the contents of the slice. This is perfect to mimic the `Body` member of an HTTP response.

We will write a test to try `response_parser`. Both methods return the same type, so we can use the same JSON parser for both:

```
func TestOpenWeatherMap_responseParser(t *testing.T) {
```

```
r := getMockData()
openWeatherMap := CurrentWeatherData{APIkey: ""}

weather, err := openWeatherMap.responseParser(r)
if err != nil {
    t.Fatal(err)
}

if weather.ID != 3117735 {
    t.Errorf("Madrid id is 3117735, not %d\n", weather.ID)
}
}
```

In the preceding test, we first asked for some mock data, which we store in the variable `r`. Later, we created a type of `CurrentWeatherData`, which we called `openWeatherMap`. Finally, we asked for a weather value for the provided `io.Reader` interface that we store in the variable `weather`. After checking for errors, we make sure that the ID is the same as the one stored in the mock data that we got from the `getMockData` method.

We have to declare the `responseParser` method before running tests, or the code won't compile:

```
func (p *CurrentWeatherData) responseParser(body io.Reader) (*Weather,
error) {
    return nil, fmt.Errorf("Not implemented yet")
}
```

With all the aforementioned, we can run this test:

```
go test -v -run=responseParser .
== RUN TestOpenWeatherMap_responseParser
--- FAIL: TestOpenWeatherMap_responseParser (0.00s)
    facade_test.go:72: Not implemented yet
FAIL
exit status 1
FAIL
```

Okay. We won't write more tests, because the rest would be merely integration tests, which are outside of the scope of explanation of a structural pattern, and will force us to have an API key as well as an Internet connection. If you want to see what the integration tests look like for this example, refer to the code that comes bundled with the book.

Implementation

First of all, we are going to implement the parser that our methods will use to parse the JSON response from the OpenWeatherMap REST API:

```
func (p *CurrentWeatherData) responseParser(body io.Reader) (*Weather, error) {
    w := new(Weather)
    err := json.NewDecoder(body).Decode(w)
    if err != nil {
        return nil, err
    }

    return w, nil
}
```

And this should be enough to pass the test by now:

```
go test -v -run=responseParser .
==== RUN    TestOpenWeatherMap_responseParser
--- PASS: TestOpenWeatherMap_responseParser (0.00s)
PASS
ok
```

At least we have our parser well tested. Let's structure our code to look like a library. First, we will create the methods to retrieve the weather of a city by its name and its country code, and the method that uses its latitude and longitude:

```
func (c *CurrentWeatherData) GetByGeoCoordinates(lat, lon float32) (*Weather, error) {
    return c.doRequest(
        fmt.Sprintf("http://api.openweathermap.org/data/2.5/weather?q=%s,%s&APPID=%s", lat, lon, c.APIkey))
}

func (c *CurrentWeatherData) GetByCityAndCountryCode(city, countryCode string) (*Weather, error) {
    return c.doRequest(
        fmt.Sprintf("http://api.openweathermap.org/data/2.5/weather?lat=%f&lon=%f&APPID=%s", city, countryCode, c.APIkey))
}
```

A piece of cake? Of course! Everything must be as easy as possible, and it is a sign of a good job. The complexity in this facade is to create connections to the OpenWeatherMap API, and control the possible errors. This problem is shared between all the Facade methods in our example, so we don't need to write more than one API call right now.

What we do is pass the URL that the REST API needs in order to return the information we desire. This is achieved by the `fmt.Sprintf` function, which formats the strings in each case. For example, to gather the data using a city name and a country code, we use the following string:

```
fmt.Sprintf("http://api.openweathermap.org/data/2.5/weather?lat=%f&lon=%f&A  
PPID=%s", city, countryCode, c.APIkey)
```

This takes the pre-formatted string `https://openweathermap.org/api` and formats it by replacing each `%s` specifier with the city, the `countryCode` that we introduced in the arguments, and the `API key` member of the `CurrentWeatherData` type.

But, we haven't set any API key! Yes, because this is a library, and the users of the library will have to use their own API keys. We are hiding the complexity of creating the URIs, and handling the errors.

Finally, the `doRequest` function is a big fish, so we will see it in detail, step by step:

```
func (o *CurrentWeatherData) doRequest(uri string) (*Weather, error) {  
    client := &http.Client{}  
    req, err := http.NewRequest("GET", uri, nil)  
    if err != nil {  
        return  
    }  
    req.Header.Set("Content-Type", "application/json")
```

First, the signature tells us that the `doRequest` method accepts a URI string, and returns a pointer to the `Weather` variable and an error. We start by creating an `http.Client` class, which will make the requests. Then, we create a request object, which will use the `GET` method, as described in the OpenWeatherMap webpage, and the URI we passed. If we were to use a different method, or more than one, they would have to be brought about by arguments in the signature. Nevertheless, we will use just the `GET` method, so we could hardcode it there.

Then, we check whether the request object has been created successfully, and set a header that says that the content type is a JSON:

```
resp, err := client.Do(req)  
if err != nil {  
    return  
}  
  
if resp.StatusCode != 200 {  
    byt, errMsg := ioutil.ReadAll(resp.Body)
```

```
if errMsg == nil {
    errMsg = fmt.Errorf("%s", string(byt))
}
err = fmt.Errorf("Status code was %d, aborting. Error message
was:\n%s\n", resp.StatusCode, errMsg)

return
}
```

Then we make the request, and check for errors. Because we have given names to our return types, if any error occurs, we just have to return the function, and Go will return the variable `err` and the variable `weather` in the state they were in at that precise moment.

We check the status code of the response, as we only accept 200 as a good response. If 200 isn't returned, we will create an error message with the contents of the body and the status code returned:

```
weather, err = o.responseParser(resp.Body)
resp.Body.Close()

return
}
```

Finally, if everything goes well, we use the `responseParser` function we wrote earlier to parse the contents of `Body`, which is an `io.Reader` interface. Maybe you are wondering why we aren't controlling `err` from the `response_parser` method. It's funny, because we are actually controlling it. `responseParser` and `doRequest` have the same return signature. Both return a `Weather` pointer and an error (if any), so we can return directly whatever the result was.

Library created with the Facade pattern

We have the first milestone for a library for the OpenWeatherMap API using the facade pattern. We have hidden the complexity of accessing the OpenWeatherMap REST API in the `doRequest` and `responseParser` functions, and the users of our library have an easy-to-use syntax to query the API. For example, to retrieve the weather for Madrid, Spain, a user will only have to introduce arguments and an API key at the beginning:

```
weatherMap := CurrentWeatherData{*apiKey}

weather, err := weatherMap.GetByCityAndCountryCode("Madrid", "ES")
if err != nil {
    t.Fatal(err)
}
```

```
fmt.Printf("Temperature in Madrid is %f celsius\n",
weather.Main.Temp-273.15)
```

The console output for the weather in Madrid at the moment of writing this chapter is the following:

```
$ Temperature in Madrid is 30.600006 celsius
```

A typical summer day!

Flyweight design pattern

Our next pattern is the **Flyweight** design pattern. It's very commonly used in computer graphics and the video game industry, but not so much in enterprise applications.

Description

Flyweight is a pattern which allows sharing the state of a heavy object between many instances of some type. Imagine that you have to create and store too many objects of some heavy type that are fundamentally equal. You'll run out of memory pretty quickly. This problem can be easily solved with the Flyweight pattern, with additional help of the Factory pattern. The factory is usually in charge of encapsulating object creation, as we saw previously.

Objectives

Thanks to the Flyweight pattern, we can share all possible states of objects in a single common object, and thus minimize object creation by using pointers to already created objects.

Example

To give an example, we are going to simulate something that you find on betting webpages. Imagine the final match of the European championship, which is viewed by millions of people across the continent. Now imagine that we own a betting webpage, where we provide historical information about every team in Europe. This is plenty of information, which is usually stored in some distributed database, and each team has, literally, megabytes of information about their players, matches, championships, and so on.

If a million users access information about a team and a new instance of the information is created for each user querying for historical data, we will run out of memory in the blink of an eye. With our Proxy solution, we could make a cache of the n most recent searches to speed up queries, but if we return a clone for every team, we will still get short on memory (but faster thanks to our cache). Funny, right?

Instead, we will store each team's information just once, and we will deliver references to them to the users. So, if we face a million users trying to access information about a match, we will actually just have two teams in memory with a million pointers to the same memory direction.

Acceptance criteria

The acceptance criteria for a Flyweight pattern must always reduce the amount of memory that is used, and must be focused primarily on this objective:

1. We will create a Team struct with some basic information such as the team's name, players, historical results, and an image depicting their shield.
2. We must ensure correct team creation (note the word *creation* here, candidate for a creational pattern), and not having duplicates.
3. When creating the same team twice, we must have two pointers pointing to the same memory address.

Basic structs and tests

Our Team struct will contain other structs inside, so a total of four structs will be created. The Team struct has the following signature:

```
type Team struct {
    ID          uint64
    Name        string
```

```
    Shield      []byte
    Players     []Player
    HistoricalData []HistoricalData
}
```

Each team has an ID, a name, some image in an slice of bytes representing the team's shield, a slice of players, and a slice of historical data. This way, we will have two teams' ID:

```
const (
    TEAM_A = iota
    TEAM_B
)
```

We declare two constants by using the `const` and `iota` keywords. The `const` keyword simply declares that the following declarations are constants. `iota` is a untyped integer that automatically increments its value for each new constant between the parentheses.

The `iota` value starts to reset to 0 when we declare `TEAM_A`, so `TEAM_A` is equal to 0. On the `TEAM_B` variable, `iota` is incremented by one so `TEAM_B` is equal to 1.

The `iota` assignment is an elegant way to save typing when declaring constant values that doesn't need specific value (like the `Pi` constant on the `math` package).

Our `Player` and `HistoricalData` are the following:

```
type Player struct {
    Name      string
    Surname   string
    PreviousTeam uint64
    Photo     []byte
}

type HistoricalData struct {
    Year          uint8
    LeagueResults []Match
}
```

As you can see, we also need a `Match` struct, which is stored within `HistoricalData` struct. A `Match` struct, in this context, represents the historical result of a match:

```
type Match struct {
    Date        time.Time
    VisitorID   uint64
    LocalID     uint64
    LocalScore   byte
    VisitorScore byte
    LocalShoots  uint16
}
```

```
    VisitorShoots uint16  
}
```

This is enough to represent a team, and to fulfill *Acceptance Criteria 1*. You have probably guessed that there is a lot of information on each team, as some of the European teams have existed for more than 100 years.

For *Acceptance Criteria 2*, the word *creation* should give us some clue about how to approach this problem. We will build a factory to create and store our teams. Our Factory will consist of a map of years, including pointers to Teams as values, and a `GetTeam` function. Using a map will boost the team search if we know their names in advance. We will also dispose of a method to return the number of created objects, which will be called the `GetNumberOfObjects` method:

```
type teamFlyweightFactory struct {  
    createdTeams map[string]*Team  
}  
  
func (t *teamFlyweightFactory) GetTeam(name string) *Team {  
    return nil  
}  
  
func (t *teamFlyweightFactory) GetNumberOfObjects() int {  
    return 0  
}
```

This is enough to write our first unit test:

```
func TestTeamFlyweightFactory_GetTeam(t *testing.T) {  
    factory := teamFlyweightFactory{}  
  
    teamA1 := factory.GetTeam(TEAM_A)  
    if teamA1 == nil {  
        t.Error("The pointer to the TEAM_A was nil")  
    }  
  
    teamA2 := factory.GetTeam(TEAM_A)  
    if teamA2 == nil {  
        t.Error("The pointer to the TEAM_A was nil")  
    }  
  
    if teamA1 != teamA2 {  
        t.Error("TEAM_A pointers weren't the same")  
    }  
  
    if factory.GetNumberOfObjects() != 1 {  
        t.Errorf("The number of objects created was not 1: %d\n",
```

```
factory.GetNumberOfObjects() )  
}  
}
```

In our test, we verify all the acceptance criteria. First we create a factory, and then ask for a pointer of TEAM_A. This pointer cannot be nil, or the test will fail.

Then we call for a second pointer to the same team. This pointer can't be nil either, and it should point to the same memory address as the previous one so we know that it has not allocated a new memory.

Finally, we should check whether the number of created teams is only one, because we have asked for the same team twice. We have two pointers but just one instance of the team. Let's run the tests:

```
$ go test -v -run=GetTeam .  
==== RUN TestTeamFlyweightFactory_GetTeam  
--- FAIL: TestTeamFlyweightFactory_GetTeam (0.00s)  
flyweight_test.go:11: The pointer to the TEAM_A was nil  
flyweight_test.go:21: The pointer to the TEAM_A was nil  
flyweight_test.go:31: The number of objects created was not 1: 0  
FAIL  
exit status 1  
FAIL
```

Well, it failed. Both pointers were nil and it has not created any object. Interestingly, the function that compares the two pointers doesn't fail; all in all, nil equals nil.

Implementation

Our GetTeam method will need to scan the map field called createdTeams to make sure the queried team is already created, and return it if so. If the team wasn't created, it will have to create it and store it in the map before returning:

```
func (t *teamFlyweightFactory) GetTeam(teamID int) *Team {  
    if t.createdTeams[teamID] != nil {  
        return t.createdTeams[teamID]  
    }  
  
    team := getTeamFactory(teamID)  
    t.createdTeams[teamID] = &team  
  
    return t.createdTeams[teamID]  
}
```

The preceding code is very simple. If the parameter name exists in the `createdTeams` map, return the pointer. Otherwise, call a factory for team creation. This is interesting enough to stop for a second and analyze. When you use the Flyweight pattern, it is very common to have a Flyweight factory, which uses other types of creational patterns to retrieve the objects it needs.

So, the `getTeamFactory` method will give us the team we are looking for, we will store it in the map, and return it. The team factory will be able to create the two teams: `TEAM_A` and `TEAM_B`:

```
func getTeamFactory(team int) Team {
    switch team {
        case TEAM_B:
            return Team{
                ID:    2,
                Name: TEAM_B,
            }
        default:
            return Team{
                ID:    1,
                Name: TEAM_A,
            }
    }
}
```

We are simplifying the objects' content so that we can focus on the Flyweight pattern's implementation. Okay, so we just have to define the function to retrieve the number of objects created, which is done as follows:

```
func (t *teamFlyweightFactory) GetNumberOfObjects() int {
    return len(t.createdTeams)
}
```

This was pretty easy. The `len` function returns the number of elements in an array or slice, the number of characters in a `string`, and so on. It seems that everything is done, and we can launch our tests again:

```
$ go test -v -run=GetTeam .
== RUN TestTeamFlyweightFactory_GetTeam
--- FAIL: TestTeamFlyweightFactory_GetTeam (0.00s)
panic: assignment to entry in nil map [recovered]
panic: assignment to entry in nil map

goroutine 5 [running]:
panic(0x530900, 0xc0820025c0)
    /home/mcastro/Go/src/runtime/panic.go:481 +0x3f4
```

```
testing.tRunner.func1(0xc082068120)
    /home/mcastro/Go/src/testing/testing.go:467 +0x199
panic(0x530900, 0xc0820025c0)
    /home/mcastro/Go/src/runtime/panic.go:443 +0x4f7
/home/mcastro/go-design-
patterns/structural/flyweight.(*teamFlyweightFactory).GetTeam(0xc08202fec0,
0x0, 0x0)
    /home/mcastro/Desktop/go-design-
patterns/structural/flyweight/flyweight.go:71 +0x159
/home/mcastro/go-design-
patterns/structural/flyweight.TestTeamFlyweightFactory_GetTeam(0xc082068120
)
    /home/mcastro/Desktop/go-design-
patterns/structural/flyweight/flyweight_test.go:9 +0x61
testing.tRunner(0xc082068120, 0x666580)
    /home/mcastro/Go/src/testing/testing.go:473 +0x9f
created by testing.RunTests
    /home/mcastro/Go/src/testing/testing.go:582 +0x899
exit status 2
FAIL
```

Panic! Have we forgotten something? By reading the stack trace on the panic message, we can see some addresses, some files, and it seems that the `GetTeam` method is trying to assign an entry to a nil map on *line 71* of the `flyweight.go` file. Let's look at *line 71* closely (remember, if you are writing code while following this tutorial, that the error will probably be in a different line so look closely at your own stark trace):

```
t.createdTeams[teamName] = &team
```

Okay, this line is on the `GetTeam` method, and, when the method passes through here, it means that it had not found the team on the map—it has created it (the variable `team`), and is trying to assign it to the map. But the map is nil, because we haven't initialized it when creating the factory. This has a quick solution. In our test, initialize the map where we have created the factory:

```
factory := teamFlyweightFactory{
    createdTeams: make(map[int]*Team, 0),
}
```

I'm sure you have seen the problem here already. If we don't have access to the package, we can initialize the variable. Well, we can make the variable public, and that's all. But this would involve every implementer necessarily knowing that they have to initialize the map, and its signature is neither convenient, or elegant. Instead, we are going to create a simple factory builder to do it for us. This is a very common approach in Go:

```
func NewTeamFactory() teamFlyweightFactory {
```

```
        return teamFlyweightFactory{
            createdTeams: make(map[int]*Team),
        }
    }
```

So now, in the test, we replace the factory creation with a call to this function:

```
func TestTeamFlyweightFactory_GetTeam(t *testing.T) {
    factory := NewTeamFactory()
    ...
}
```

And we run the test again:

```
$ go test -v -run=GetTeam .
== RUN TestTeamFlyweightFactory_GetTeam
--- PASS: TestTeamFlyweightFactory_GetTeam (0.00s)
PASS
ok
```

Perfect! Let's improve the test by adding a second test, just to ensure that everything will be running as expected with more volume. We are going to create a million calls to the team creation, representing a million calls from users. Then, we will simply check that the number of teams created is only two:

```
func Test_HighVolume(t *testing.T) {
    factory := NewTeamFactory()

    teams := make([]*Team, 500000*2)
    for i := 0; i < 500000; i++ {
        teams[i] = factory.GetTeam(TEAM_A)
    }

    for i := 500000; i < 2*500000; i++ {
        teams[i] = factory.GetTeam(TEAM_B)
    }

    if factory.GetNumberOfObjects() != 2 {
        t.Errorf("The number of objects created was not 2:
%d\n", factory.GetNumberOfObjects())
    }
}
```

In this test, we retrieve TEAM_A and TEAM_B 500,000 times each to reach a million users. Then, we make sure that just two objects were created:

```
$ go test -v -run=Volume .
== RUN Test_HighVolume
```

```
---- PASS: Test_HighVolume (0.04s)
PASS
ok
```

Perfect! We can even check where the pointers are pointing to, and where they are located. We will check with the first three as an example. Add these lines at the end of the last test, and run it again:

```
for i:=0; i<3; i++ {
    fmt.Printf("Pointer %d points to %p and is located in %p\n", i, teams[i],
    &teams[i])
}
```

In the preceding test, we use the `Printf` method to print information about pointers. The `%p` flag gives you the memory location of the object that the pointer is pointing to. If you reference the pointer by passing the `&` symbol, it will give you the direction of the pointer itself.

Run the test again with the same command; you will see three new lines in the output with information similar to the following:

```
Pointer 0 points to 0xc082846000 and is located in 0xc082076000
Pointer 1 points to 0xc082846000 and is located in 0xc082076008
Pointer 2 points to 0xc082846000 and is located in 0xc082076010
```

What it tells us is that the first three positions in the map point to the same location, but that we actually have three different pointers, which are, effectively, much lighter than our team object.

What's the difference between Singleton and Flyweight then?

Well, the difference is subtle but it's just there. With the Singleton pattern, we ensure that the same type is created only once. Also, the Singleton pattern is a Creational pattern. With Flyweight, which is a Structural pattern, we aren't worried about how the objects are created, but about how to structure a type to contain heavy information in a light way. The structure we are talking about is the `map[int]*Team` structure in our example. Here, we really didn't care about how we created the object; we have simply written an uncomplicated `getTeamFactory` method for it. We gave major importance to having a light structure to hold a shareable object (or objects), in this case, the map.

Summary

We have seen several patterns to organize code structures. Structural patterns are concerned about how to create objects, or how they do their business (we'll see this in the behavioral patterns).

Don't feel confused about mixing several patterns. You could end up mixing six or seven quite easily if you strictly follow the objectives of each one. Just keep in mind that over-engineering is as bad as no engineering at all. I remember prototyping a load balancer one evening, and after two hours of crazy over-engineered code, I had such a mess in my head that I preferred to start all over again.

In the next chapter, we'll see behavioral patterns. They are a bit more complex, and they often use Structural and Creational patterns for their objectives, but I'm sure that the reader will find them quite challenging and interesting.

5

Behavioral Patterns - Strategy, Chain of Responsibility, and Command Design Patterns

The last group of common patterns we are going to see are the behavioral patterns. Now, we aren't going to define structures or encapsulate object creation but we are going to deal with behaviors.

What's to deal with in behavior patterns? Well, now we will encapsulate behaviors, for example, algorithms in the Strategy pattern or executions in the command pattern.

Correct Behavior design is the last step after knowing how to deal with object creation and structures. Defining the behavior correctly is the last step of good software design because, all in all, good software design lets us improve algorithms and fix errors easily while the best algorithm implementation will not save us from bad software design.

Strategy design pattern

The Strategy pattern is probably the easiest to understand of the Behavioral patterns. We have used it a few times while developing the previous patterns but without stopping to talk about it. Now we will.

Description

The Strategy pattern uses different algorithms to achieve some specific functionality. These algorithms are hidden behind an interface and, of course, they must be interchangeable. All algorithms achieve the same functionality in a different way. For example, we could have a Sort interface and few sorting algorithms. The result is the same, some list is sorted, but we could have used quick sort, merge sort, and so on.

Can you guess when we used a Strategy pattern in the previous chapters? Three, two, one... Well, we heavily used the strategy pattern when we used the `io.Writer` interface. The `io.Writer` interface defines a strategy to write, and the functionality is always the same--to write something. We could write it to the standard out, to some file or to a user-defined type, but we do the same thing at the end--to write. We just change the strategy to write (in this case, we change the place where we write).

Objectives

The objectives of the Strategy pattern are really clear. The pattern should do the following:

- Provide a few algorithms to achieve some specific functionality
- All types achieve the same functionality in a different way but the client of the strategy isn't affected

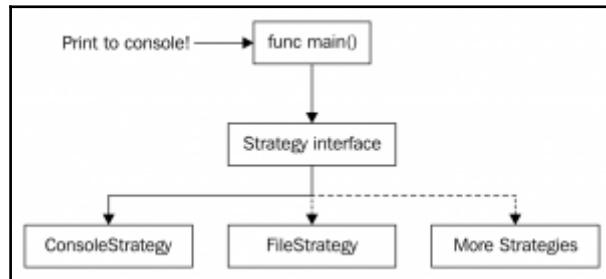
The problem is that this definition covers a huge spectrum of possibilities. This is because Strategy pattern is actually used for a variety of scenarios and many software engineering solutions come with some kind of strategy within. Therefore it's better to see it in action with a real example.

Rendering images or text

We are going to do something different for this example. Instead of printing text on the console only, we are also going to paint objects on a file.

In this case, we will have two strategies: console and file. But the user of the library won't have to deal with the complexity behind them.

The key feature is that the "caller" doesn't know how the underlying library is working and he just knows the information available on the defined strategy. This is nicely seen on the following diagram:



In this diagram, we have chosen to print to console but we won't deal with the **ConsoleStrategy** type directly, we'll always use an interface that represents it. The **ConsoleStrategy** type will hide the implementation details of printing to console to caller in `main` function. **FileStrategy** hides its implementation details as well as any future strategy.

Acceptance criteria

A strategy must have a very clear objective and we will have two ways to achieve it. Our objectives will be as follows:

- Provide a way to show to the user an object (a square) in text or image
- The user must choose between image or text when launching the app
- The app must be able to add more visualization strategies (audio, for example)
- If the user selects text, the word *Square* must be printed in the console
- If the user selects image, an image of a white square on a black background will be printed on a file

Implementation

We aren't going to write tests for this example as it will be quite complicated to check that an image has appeared on the screen (although not impossible by using **OpenCV**, an impressive library for computer vision). We will start directly by defining our strategy interface that each printing strategy must implement (in our case, the file and console types):

```
type PrintStrategy interface {
    Print() error
}
```

That's all. Our strategy defines a simple `Print()` method that returns an `error` (the error-returning type is mandatory when dealing with files, for example). The types that needs to implement `PrintStrategy` will be called `ConsoleSquare` and a `ImageSquare` type:

```
type ConsoleSquare struct {}

type ImageSquare struct {
    DestinationFilePath string
}
```

The `ConsoleSquare` struct doesn't need any inner field because it will always print the word `Square` to the console. The `ImageSquare` struct will store a field for the destination of the image file where we will print the square. We will start with the implementation of the `ConsoleSquare` type as it is the simplest:

```
func(c *ConsoleSquare) Print() error {
    println("Square")
    return nil
}
```

Very easy, but the image is more complex. We won't spend too much time in explaining in detail how the `image` package works because the code is easily understandable:

```
func (t *ImageSquare) Print() error {
    width := 800
    height := 600

    origin := image.Point{0, 0}

    bgImage := image.NewRGBA(image.Rectangle{
        Min: origin,
        Max: image.Point{X: width, Y: height},
    })
}
```

```
bgColor := image.Uniform{color.RGBA{R: 70, G: 70, B: 70, A:0}}
quality := &jpeg.Options{Quality: 75}

draw.Print(bgImage, bgImage.Bounds(), &bgColor, origin, draw.Src)
```

However, here is a short explanation:

- We define a size for the image (`width` and `height` variables) of 800 pixels of width and 600 pixels of height. Those are going to be the size limits of our image and anything that we write outside of that size won't be visible.
- The `origin` variable stores an `image.Point`, a type to represent a position in any two-dimensional space. We set the position of this point at $(0, 0)$, the upper-left corner of the image.
- We need a bitmap that will represent our background, here we called it `bgImage`. We have a very handy function in the `image` package to create the `image.RGBA` types called `image.NewRGBA`. We need to pass a rectangle to this function so that it knows the bounds of the image. A rectangle is represented by two `image.Point` types--its upper left corner point (the `Min` field) and its lower right corner point (the `Max` field). We use `origin` as the upper-left and a new point with the values of `width` and `height` as the lower-right point.
- The image will have a gray background color (`bgColor`). This is done by instancing a type of `image.Uniform`, which represents a uniform color (hence the name). The `image.Uniform` type needs an instance of a `color.Color` interface. A `color.Color` type is any type that implements the `RGB()` (`r, g, b, a uint32`) method to return a `uint32` value for red, green, blue, and alpha colors (`RGBA`). Alpha is a value for the transparency of a pixel. The `color` package conveniently provides a type called `color.RGBA` for this purpose (in case we don't need to implement our own, which is our case).
- When storing an image in certain formats, we have to specify the quality of the image. It will affect not only the quality but the size of the file, of course. Here, it is defined as 75; 100 is the maximum quality possible that we can set. As you can see, we are using the `jpeg` package here to set the value of a type called `Options` that simply stores the value of the quality, it doesn't have more values to apply.

- Finally, the `draw.Print` function writes the pixels on the supplied image (`bgImage`) with the characteristics that we have defined on the bounds defined by the same image. The first argument of the `draw.Print` method takes the destination image, where we used `bgImage`. The second argument is the bounds of the object to draw in the destination image, we used the same bounds of the image but we could use any other if we wanted a smaller rectangle. The third argument is the color to use to colorize the bounds. The `origin` variable is used to tell where the upper-left corner of the bound must be placed. In this case, the bounds are the same size as the image so we need to set it to the origin. The last argument specified is the operation type; just leave it in the `draw.Src` argument.

Now we have to draw the square. The operation is essentially the same as to draw the background but, in this case, we are drawing a square over the previously drawn `bgImage`:

```
squareWidth := 200
squareHeight := 200
squareColor := image.Uniform{color.RGBA{R: 255, G: 0, B: 0, A: 1}}
square := image.Rect(0, 0, squareWidth, squareHeight)
square = square.Add(image.Point{
    X: (width / 2) - (squareWidth / 2),
    Y: (height / 2) - (squareHeight / 2),
})
squareImg := image.NewRGBA(square)

draw.Print(bgImage, squareImg.Bounds(), &squareColor, origin, draw.Src)
```

The square will be of 200*200 pixels of red color. When using the method `Add`, the `Rect` type `origin` is translated to the supplied point; this is to center the square on the image. We create an image with the square `Rect` and call the `Print` function on the `bgImage` image again to draw the red square over it:

```
w, err := os.Create(t.DestinationFilePath)
if err != nil {
    return fmt.Errorf("Error opening image")
}
defer w.Close()

if err = jpeg.Encode(w, bgImage, quality); err != nil {
    return fmt.Errorf("Error writing image to disk")
}

return nil
}
```

Finally, we will create a file to store the contents of the image. The file will be stored in the path supplied in the `DestinationFilePath` field of the `ImageSquare` struct. To create a file, we use `os.Create` that returns the `*os.File`. As with every file, it must be closed after using it so don't forget to use the `defer` keyword to ensure that you close it when the method finishes.



To defer, or not to defer? Some people ask why the use of `defer` at all? Wouldn't it be the same to simply write it without `defer` at the end of the function? Well, actually not. If any error occurs during the method execution and you return this error, the `Close` method won't be executed if it's at the end of the function. You can close the file before returning but you'll have to do it in every error check. With `defer`, you don't have to worry about this because the deferred function is executed always (with or without error). This way, we ensure that the file is closed.

To parse the arguments, we'll use the `flag` package. We have used it before but let's recall its usage. A flag is a command that the user can pass when executing our app. We can define a flag by using the `flag.[type]` methods defined in the `flag` package. We want to read the output that the user wants to use from the console. This flag will be called `output`. A flag can have a default value; in this case, it will have the value `console` that will be used when printing to console. So, if the user executes the program without arguments, it prints to `console`:

```
var output = flag.String("output", "console", "The output to use between  
'console' and 'image' file")
```

Our final step is to write the main function:

```
func main() {  
    flag.Parse()
```

Remember that the first thing to do in the main when using flags is to parse them using the `flag.Parse()` method! It's very common to forget this step:

```
var activeStrategy PrintStrategy

switch *output {
case "console":
    activeStrategy = &TextSquare{}
case "image":
    activeStrategy = &ImageSquare{/tmp/image.jpg}
default:
    activeStrategy = &TextSquare{}
}
```

We define a variable for the strategy that the user has chosen, called `activeStrategy`. But check that the `activeStrategy` variable has the `PrintStrategy` type so it can be populated with any implementation of the `PrintStrategy` variable. We will set `activeStrategy` to a new instance of `TextSquare` when the user writes the `--output=console` command and an `ImageSquare` when we write the `--output=image` command.

Finally, here is the design pattern execution:

```
err := activeStrategy.Print()
if err != nil {
    log.Fatal(err)
}
}
```

Our `activeStrategy` variable is a type implementing `PrintStrategy` and either the `TextSquare` or `ImageSquare` classes. The user will choose at runtime which strategy he wants to use for each particular case. Also, we could have written a factory method pattern to create strategies, so that the strategy creation will also be uncoupled from the main function and abstracted in a different independent package. Think about it: if we have the strategy creation in a different package, it will also allow us to use this project as a library and not only as a standalone app.

Now we will execute both strategies; the `TextSquare` instance will give us a square by printing the word `Square` on the console:

```
$ go run main.go --output=console
Square
```

It has worked as expected. Recalling how flags work, we have to use the `--` (double dash) and the defined flag, `output` in our case. Then you have two options--using `=` (equals) and immediately writing the value for the flag or writing `<space>` and the value for the flag. In this case, we have defined the default value of `output` to the console so the following three executions are equivalent:

```
$ go run main.go --output=console
Square
$ go run main.go --output console
Square
$ go run main.go
Square
```

Now we have to try the file strategy. As defined before, the file strategy will print a red square to a file as an image with dark gray background:

```
$ go run main.go --output image
```

Nothing happened? But everything worked correctly. This is actually bad practice. Users must always have some sort of feedback when using your app or your library. Also, if they are using your code as a library, maybe they have a specific format for output so it won't be nice to directly print to the console. We will solve this issue later. Right now, open the folder `/tmp` with your favourite file explorer and you will see a file called `image.jpg` with our red square in a dark grey background.

Solving small issues in our library

We have a few issues in our code:

- It cannot be used as a library. We have critical code written in the `main` package (strategy creation). **Solution:** Abstract to two different packages the strategy creation from the command-line application.
- None of the strategies are doing any logging to file or console. We must provide a way to read some logs that an external user can integrate in their logging strategies or formats. **Solution:** Inject an `io.Writer` interface as dependency to act as a logging sink.

- Our `TextSquare` class is always writing to the console (an implementer of the `io.Writer` interface) and the `ImageSquare` is always writing to file (another implementer of the `io.Writer` interface). This is too coupled. **Solution:** Inject an `io.Writer` interface so that the `TextSquare` and `ImageSquare` can write to any of the `io.Writer` implementations that are available (file and console, but also bytes buffer, binary encoders, JSON handlers... dozens of packages).

So, to use it as a library and solve the first issue, we will follow a common approach in Go file structures for apps and libraries. First, we will place our main package and function outside of the root package; in this case, in a folder called `cli`. It is also common to call this folder `cmd` or even `app`. Then, we will place our `PrintStrategy` interface in the root package, which now will be called the `strategy` package. Finally, we will create a `shapes` package in a folder with the same name where we will put both text and image strategies. So, our file structure will be like this:

- **Root package:** `strategy`

File: `print_strategy.go`

- **SubPackage:** `shapes`

Files: `image.go`, `text.go`, `factory.go`

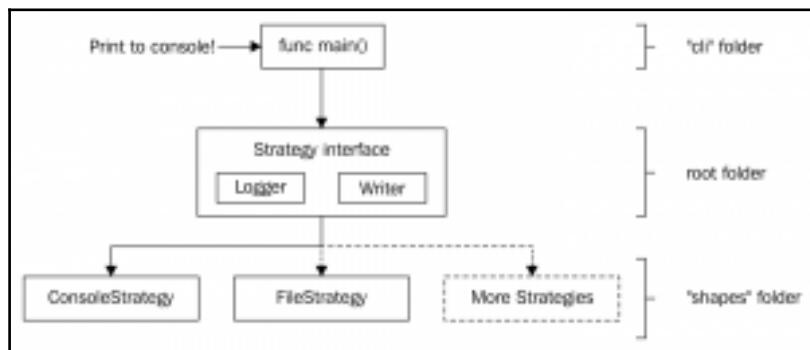
- **SubPackage:** `cli`

File: `main.go`

We are going to modify our interface a bit to fit the needs we have written previously:

```
type PrintStrategy interface {
    Print() error
    SetLog(io.Writer)
    SetWriter(io.Writer)
}
```

We have added the `SetLog(io.Writer)` method to add a logger strategy to our types; this is to provide feedback to users. Also, it has a `SetWriter` method to set the `io.Writer` strategy. This interface is going to be located on the root package in the `print_strategy.go` file. So the final schema looks like this:



Both the `TextSquare` and `ImageSquare` strategies have to satisfy the `SetLog` and `SetWriter` methods which simply store some object on their fields so, instead of implementing the same twice, we can create a struct that implements them and embed this struct in the strategies. By the way, this would be the composite pattern we have seen previously:

```
type PrintOutput struct {
    Writer     io.Writer
    LogWriter  io.Writer
}

func(d *PrintOutput) SetLog(w io.Writer) {
    d.LogWriter = w
}

func(d *PrintOutput) SetWriter(w io.Writer) {
    d.Writer = w
}
```

So now each strategy must have the `PrintOutput` struct embedded if we want to modify their `Writer` and `logger` fields.

We also need to modify our strategy implementation. The `TextSquare` struct now needs a field to store the output `io.Writer` (the place where it is going to write instead of writing always to the console) and the `log` writer. These two fields can be provided by embedding the `PrintOutput` struct. The `TextSquare` struct is also stored in the file `text.go` within the `shapes` package. So, the struct is now like this:

```
package shapes

type TextSquare struct {
    strategy.PrintOutput
}
```

So now the `Print()` method is slightly different because, instead of writing directly to the console by using the `println` function, we have to write whichever `io.Writer` is stored in the `Writer` field:

```
func (t *TextSquare) Print() error {
    r := bytes.NewReader([]byte("Circle"))
    io.Copy(t.Writer, r)
    return nil
}
```

The `bytes.NewReader` is a very useful function that takes an array of bytes and converts them to an `io.Reader` interface. We need an `io.Reader` interface to use the `io.Copy` function. The `io.Copy` function is also incredibly useful as it takes an `io.Reader` (as the second parameter) and pipes it to an `io.Writer` (its first parameter). So, we won't return an error in any case. However, it's easier to do so using directly the `Write` method of `t.Writer`:

```
func (t *TextSquare) Print() error {
    t.Writer.Write([]byte("Circle"))
    return nil
}
```

You can use whichever method you like more. Usually, you will use the `Write` method but it's nice to know the `bytes.NewReader` function too.

Did you realize that when we use `t.Writer`, we are actually accessing `PrintOutput.Writer`? The `TextSquare` type has a `Writer` field because the `PrintOutput` struct has it and it's embedded on the `TextSquare` struct.



Embedding is not inheritance. We have embedded the `PrintOutput` struct on the `TextSquare` struct. Now we can access `PrintOutput` fields as if they were in `TextSquare` fields. This feels a bit like inheritance but there is a very important difference here: `TextSquare` is not a `PrintOutput` value but it has a `PrintOutput` in its composition. What does it mean? That if you have a function that expects a `PrintOutput`, you cannot pass `TextSquare` just because it has a `PrintOutput` embedded. But, if you have a function that accepts an interface that `PrintOutput` implements, you can pass `TextSquare` if it has a `PrintOutput` embedded. This is what we are doing in our example.

The `ImageSquare` struct is now like the `TextSquare`, with a `PrintOutput` embedded:

```
type ImageSquare struct {
    strategy.PrintOutput
}
```

The `Print` method also needs to be modified. Now, we aren't creating a file from the `Print` method, as it was breaking the single responsibility principle. A file implements an `io.Writer` so we will open the file outside of the `ImageSquare` struct and inject it on the `Writer` field. So, we just need to modify the end of the `Print()` method where we wrote to the file:

```
draw.Print(bgImage, squareImg.Bounds(), &squareColor, origin, draw.Src)

if i.Writer == nil {
    return fmt.Errorf("No writer stored on ImageSquare")
}
if err := jpeg.Encode(i.Writer, bgImage, quality); err != nil {
    return fmt.Errorf("Error writing image to disk")
}

if i.LogWriter != nil {
    io.Copy(i.LogWriter, "Image written in provided writer\n")
}

return nil
```

If you check our previous implementation, after using `draw`, you can see that we used the `Print` method, we created a file with `os.Create` and passed it to the `jpeg.Encode` function. We have deleted this part about creating the file and we have replaced it with a check looking for a `Writer` in the fields (`if i.Writer != nil`). Then, on `jpeg.Encode` we can replace the file value we were using previously with the content of the `i.Writer` field. Finally, we are using `io.Copy` again to log some message to the `LogWriter` if a logging strategy is provided.

We also have to abstract the knowledge needed from the user to create instances of implementors of the `PrintStrategy` for which we are going to use a Factory method:

```
const (
    TEXT_STRATEGY = "text"
    IMAGE_STRATEGY = "image"
)

func NewPrinter(s string) (strategy.Output, error) {
    switch s {
    case TEXT_STRATEGY:
        return &TextSquare{
            PrintOutput: strategy.PrintOutput{
                LogWriter: os.Stdout,
            },
            nil
    case IMAGE_STRATEGY:
        return &ImageSquare{
            PrintOutput: strategy.PrintOutput{
                LogWriter: os.Stdout,
            },
            nil
    default:
        return nil, fmt.Errorf("Strategy '%s' not found\n", s)
    }
}
```

We have two constants, one of each of our strategies: `TEXT_STRATEGY` and `IMAGE_STRATEGY`. Those are the constants that must be provided to the factory to retrieve each square drawer strategy. Our factory method receives an argument `s`, which is a string with one of the previous constants.

Each strategy has a `PrintOutput` type embedded with a default logger to `stdout` but you can override it later by using the `SetLog(io.Writer)` methods. This approach could be considered a Factory of prototypes. If it is not a recognized strategy, a proper message error will be returned.

What we have now is a library. We have all the functionality we need between the strategy and shapes packages. Now we will write the main package and function in a new folder called cli:

```
var output = flag.String("output", "text", "The output to use between "+  
    "'console' and 'image' file")  
  
func main() {  
    flag.Parse()
```

Again, like before, the main function starts by parsing the input arguments on the console to gather the chosen strategy. We can use the variable output now to create a strategy without Factory:

```
activeStrategy, err := shapes.NewPrinter(*output)  
if err != nil {  
    log.Fatal(err)  
}
```

With this snippet, we have our strategy or we stop program execution in the log.Fatal method if any error is found (such as an unrecognized strategy).

Now we will implement the business needs by using our library. For the purpose of the TextStrategy, we want to write, for example, to stdout. For the purpose of the image, we will write to /tmp/image.jpg. Just like before. So, following the previous statements, we can write:

```
switch *output {  
case shapes.TEXT_STRATEGY:  
    activeStrategy.SetWriter(os.Stdout)  
case shapes.IMAGE_STRATEGY:  
    w, err := os.Create("/tmp/image.jpg")  
    if err != nil {  
        log.Fatal("Error opening image")  
    }  
    defer w.Close()  
  
    activeStrategy.SetWriter(w)  
}
```

In the case of `TEXT_STRATEGY`, we use `SetWriter` to set the `io.Writer` to `os.Stdout`. In the case of `IMAGE_STRATEGY`, we create an image in any of our folders and pass the file variable to the `SetWriter` method. Remember that `os.File` implements the `io.Reader` and `io.Writer` interfaces, so it's perfectly legal to pass it as an `io.Writer` to the `SetWriter` method:

```
err = activeStrategy.Print()
if err != nil {
    log.Fatal(err)
}
```

Finally, we call the `Print` method of whichever strategy was chosen by the user and check for possible errors. Let's try the program now:

```
$ go run main.go --output text
Circle
```

It has worked as expected. What about the image strategy?

```
$ go run main.go --output image
Image written in provided writer
```

If we check in `/tmp/image.jpg`, we can find our red square on the dark background.

Final words on the Strategy pattern

We have learned a powerful way to encapsulate algorithms in different structs. We have also used embedding instead of inheritance to provide cross-functionality between types, which will come in handy very often in our apps. You'll find yourself combining strategies here and there as we have seen in the second example, where we have strategies for logging and writing by using the `io.Writer` interface, a strategy for byte-streaming operations.

Chain of responsibility design pattern

Our next pattern is called **chain of responsibility**. As its name implies, it consists of a chain and, in our case, each link of the chain follows the single responsibility principle.

Description

The single responsibility principle implies that a type, function, method, or any similar abstraction must have one single responsibility only and it must do it quite well. This way, we can apply many functions that achieve one specific thing each to some struct, slice, map, and so on.

When we apply many of these abstractions in a logical way very often, we can chain them to execute in order such as, for example, a logging chain.

A logging chain is a set of types that logs the output of some program to more than one `io.Writer` interface. We could have a type that logs to the console, a type that logs to a file, and a type that logs to a remote server. You can make three calls every time you want to do some logging, but it's more elegant to make only one and provoke a chain reaction.

But also, we could have a chain of checks and, in case one of them fails, break the chain and return something. This is the authentication and authorization middleware works.

Objectives

The objective of the chain of responsibility is to provide to the developer a way to chain actions at runtime. The actions are chained to each other and each link will execute some action and pass the request to the next link (or not). The following are the objectives followed by this pattern:

- Dynamically chain the actions at runtime based on some input
- Pass a request through a chain of processors until one of them can process it, in which case the chain could be stopped

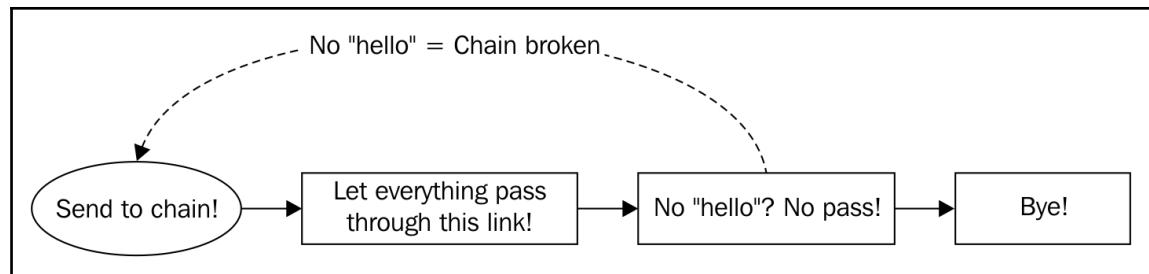
A multi-logger chain

We are going to develop a multi-logger solution that we can chain in the way we want. We will use two different console loggers and one general-purpose logger:

1. We need a simple logger that logs the text of a request with a prefix *First logger* and passes it to the next link in the chain.
2. A second logger will write on the console if the incoming text has the word `hello` and pass the request to a third logger. But, if not, the chain will be broken and it will return immediately.

3. A third logger type is a general purpose logger called `WriterLogger` that uses an `io.Writer` interface to log.
4. A concrete implementation of the `WriterLogger` writes to a file and represents the third link in the chain.

The implementation of these steps is described in the following figure:



Unit test

The very first thing to do for the chain is, as usual, to define the interface. A chain of responsibility interface will usually have, at least, a `Next()` method. The `Next()` method is the one that executes the next link in the chain, of course:

```
type ChainLogger interface {
    Next(string)
}
```

The `Next` method on our example's interface takes the message we want to log and passes it to the following link in the chain. As written in the acceptance criteria, we need three loggers:

```
type FirstLogger struct {
    NextChain ChainLogger
}

func (f *FirstLogger) Next(s string) {}

type SecondLogger struct {
    NextChain ChainLogger
}

func (f *SecondLogger) Next(s string) {}

type WriterLogger struct {
```

```
    NextChain ChainLogger
    Writer     io.Writer
}
func (w *WriterLogger) Next(s string) {}
```

The `FirstLogger` and `SecondLogger` types have exactly the same structure--both implement `ChainLogger` and have a `NextChain` field that points to the next `ChainLogger`. The `WriterLogger` type is equal to the `FirstLogger` and `SecondLogger` types but also has a field to write its data to, so you can pass any `io.Writer` interface to it.

As we have done before, we'll implement an `io.Writer` struct to use in our testing. In our test file, we define the following struct:

```
type myTestWriter struct {
    receivedMessage string
}

func (m *myTestWriter) Write(p []byte) (int, error) {
    m.receivedMessage += string(p)
    return len(p), nil
}

func(m *myTestWriter) Next(s string){
    m.Write([]byte(s))
}
```

We will pass an instance of the `myTestWriter` struct to `WriterLogger` so we can track what's being logged on testing. The `myTestWriter` class implements the common `Write([]byte) (int, error)` method from the `io.Writer` interface. Remember, if it has the `Write` method, it can be used as `io.Writer`. The `Write` method simply stored the `string` argument to the `receivedMessage` field so we can check later its value on tests.

This is the beginning of the first test function:

```
func TestCreateDefaultChain(t *testing.T) {
    //Our test ChainLogger
    myWriter := myTestWriter{}

    writerLogger := WriterLogger{Writer: &myWriter}
    second := SecondLogger{NextChain: &writerLogger}
    chain := FirstLogger{NextChain: &second}
```

Let's describe these few lines a bit as they are quite important. We create a variable with a default `myTestWriter` type that we'll use as an `io.Writer` interface in the last link of our chain. Then we create the last piece of the link chain, the `writerLogger` interface. When implementing the chain, you usually start with the last piece on the link and, in our case, it is a `WriterLogger`. The `WriterLogger` writes to an `io.Writer` so we pass `myWriter` as `io.Writer` interface.

Then we have created a `SecondLogger`, the middle link in our chain, with a pointer to the `writerLogger`. As we mentioned before, `SecondLogger` just logs and passes the message in case it contains the word `hello`. In a production app, it could be an error-only logger.

Finally, the first link in the chain has the variable name `chain`. It points to the second logger. So, to resume, our chain looks like this: `FirstLogger | SecondLogger | WriterLogger`.

This is going to be our default setup for our tests:

```
t.Run("3 loggers, 2 of them writes to console, second only if it finds " +
    "the word 'hello', third writes to some variable if second found
    'hello'",
    func(t *testing.T) {
        chain.Next("message that breaks the chain\n")

        if myWriter.receivedMessage != "" {
            t.Fatal("Last link should not receive any message")
        }

        chain.Next("Hello\n")

        if !strings.Contains(myWriter.receivedMessage, "Hello") {
            t.Fatal("Last link didn't received expected message")
        }
    })
})
```

Continuing with Go 1.7 or later testing signatures, we define an inner test with the following description: *three loggers, two of them write to console, the second only if it finds the word 'hello', the third writes to some variable if the second found 'hello'*. It's quite descriptive and very easy to understand if someone else has to maintain this code.

First, we use a message on the `Next` method that will not reach the third link in the chain as it doesn't contain the word `hello`. We check the contents of the `receivedMessage` variable, that by default is empty, to see if it has changed because it shouldn't.

Next, we use the chain variable again, our first link in the chain, and pass the message "Hello\n". According to the description of the test, it should log using `FirstLogger`, then in `SecondLogger` and finally in `WriterLogger` because it contains the word `hello` and the `SecondLogger` will let it pass.

The test checks that `myWriter`, the last link in the chain that stored the past message in a variable called `receivedMessage`, has the word that we passed first in the chain: `hello`. Let's run it so it fails:

```
go test -v .
==== RUN    TestCreateDefaultChain
==== RUN
TestCreateDefaultChain/3_loggers,_2_of_them_writes_to_console,_second_only_if_it_founds_the_word_'hello',_thirdWritesToSomeVariableIfSecondFoundHello'
--- FAIL: TestCreateDefaultChain (0.00s)
--- FAIL:
TestCreateDefaultChain/3_loggers,_2_of_them_writes_to_console,_second_only_if_it_founds_the_word_'hello',_thirdWritesToSomeVariableIfSecondFoundHello' (0.00s)
        chain_test.go:33: Last message didn't received expected message
FAIL
exit status 1
FAIL
```

The test passed for the first check of the test and didn't for the second check. Well... ideally no check should pass before any implementation is done. Remember that in test-driven development, tests must fail on the first launch because the code they are testing isn't implemented yet. Go zero-initialization misleads us with this passed check on the test. We can solve this in two ways:

- Making the signature of the `ChainLogger` to return an error: `Next (string) error`. This way, we would break the chain returning an error. This is a much more convenient way in general, but it will introduce quite a lot of boilerplate right now.
- Changing the `receivedMessage` field to a pointer. A default value of a pointer is `nil`, instead of an empty string.

We will use the second option now, as it's much simpler and quite effective too. So let's change the signature of the `myTestWriter` struct to the following:

```
type myTestWriter struct {
    receivedMessage *string
}
```

```
func (m *myTestWriter) Write(p []byte) (int, error) {
    if m.receivedMessage == nil {
        m.receivedMessage = new(string)
    }
    tempMessage := fmt.Sprintf("%s%s", m.receivedMessage, p)
    m.receivedMessage = &tempMessage
    return len(p), nil
}

func (m *myTestWriter) Next(s string) {
    m.Write([]byte(s))
}
```

Check that the type of `receivedMessage` has the asterisk (*) now to indicate that it's a pointer to a string. The `Write` function needed to change too. Now we have to check the contents of the `receivedMessage` field because, as every pointer, it's initialized to `nil`. Then we have to store the message in a variable first, so we can take the address in the next line on the assignment (`m.receivedMessage = &tempMessage`).

So now our test code should change a bit too:

```
t.Run("3 loggers, 2 of them writes to console, second only if it finds "+  
"the word 'hello', third writes to some variable if second found 'hello'",  
func(t *testing.T) {  
    chain.Next("message that breaks the chain\n")  
  
    if myWriter.receivedMessage != nil {  
        t.Error("Last link should not receive any message")  
    }  
  
    chain.Next("Hello\n")  
  
    if myWriter.receivedMessage == "" ||  
    !strings.Contains(*myWriter.receivedMessage, "Hello") {  
        t.Fatal("Last link didn't received expected message")  
    }  
})
```

Now we are checking that `myWriter.receivedMessage` is actually `nil`, so no content has been written for sure on the variable. Also, we have to change the second if to check first that the member isn't `nil` before checking its contents or it can throw a panic on test. Let's test it again:

```
go test -v .  
== RUN TestCreateDefaultChain  
== RUN  
TestCreateDefaultChain/3_loggers,_2_of_themWrites_to_console,_second_only_
```

```
if_it_founds_the_word_'hello',_thirdWritesToSomeVariableIfSecondFound_
d_'hello'
--- FAIL: TestCreateDefaultChain (0.00s)
--- FAIL:
TestCreateDefaultChain/3_loggers,_2_of_them_writes_to_console,_second_only_
if_it_founds_the_word_'hello',_thirdWritesToSomeVariableIfSecondFound_
d_'hello' (0.00s)
    chain_test.go:40: Last link didn't receive expected message
FAIL
exit status 1
FAIL
```

It fails again and, again, the first half of the test passes correctly without implemented code. So what should we do now? We have change the signature of the `myWriter` type to make the test fail in both checks and, again, just fail in the second. Well, in this case we can pass this small issue. When writing tests, we must be very careful to not get too crazy about them; unit tests are tools to help us write and maintain code, but our target is to write functionality, not tests. This is important to keep in mind as you can get really crazy engineering unit tests.

Implementation

Now we have to implement the first, second, and third loggers called `FirstLogger`, `SecondLogger`, and `WriterLogger` respectively. The `FirstLogger` logger is the easiest one as described in the first acceptance criterion: *We need a simple logger that logs the text of a request with a prefix First logger: and passes it to the next link in the chain.* So let's do it:

```
type FirstLogger struct {
    NextChain ChainLogger
}

func (f *FirstLogger) Next(s string) {
    fmt.Printf("First logger: %s\n", s)

    if f.NextChain != nil {
        f.NextChain.Next(s)
    }
}
```

The implementation is quite easy. Using the `fmt.Printf` method to format and print the incoming string, we appended the text `First Logger: text`. Then, we check that the `NextChain` type has actually some content and pass the control to it by calling its `Next(string)` method. The test shouldn't pass yet so we'll continue with the `SecondLogger` logger:

```
type SecondLogger struct {
    NextChain ChainLogger
}

func (se *SecondLogger) Next(s string) {
    if strings.Contains(strings.ToLower(s), "hello") {
        fmt.Printf("Second logger: %s\n", s)

        if se.NextChain != nil {
            se.NextChain.Next(s)
        }

        return
    }

    fmt.Printf("Finishing in second logging\n\n")
}
```

As mentioned in the second acceptance criterion, the `SecondLogger` description is: *A second logger will write on the console if the incoming text has the word "hello" and pass the request to a third logger.* First of all, it checks whether the incoming text contains the text `hello`. If it's true, it prints the message to the console, appending the text `Second logger:` and passes the message to the next link in the chain (check previous instance that a third link exists).

But if it doesn't contain the text `hello`, the chain is broken and it prints the message `Finishing in second logging`.

We'll finalize with the `WriterLogger` type:

```
type WriterLogger struct {
    NextChain ChainLogger
    Writer     io.Writer
}

func (w *WriterLogger) Next(s string) {
    if w.Writer != nil {
        w.Writer.Write([]byte("WriterLogger: " + s))
    }

    if w.NextChain != nil {
```

```
w.NextChain.Next(s)
}
}
```

The WriterLogger struct's Next method checks that there is an existing `io.Writer` interface stored in the `Writer` member and writes there the incoming message appending the text `WriterLogger:` to it. Then, like the previous links, check that there are more links to pass the message.

Now the tests will pass successfully:

```
go test -v .
==== RUN    TestCreateDefaultChain
==== RUN
TestCreateDefaultChain/3_loggers,_2_of_them_writes_to_console,_second_only_if_it_founds_the_word_'hello',_thirdWrites_to_some_variable_if_second_found_'hello'
First logger: message that breaks the chain
Finishing in second logging

First logger: Hello
Second logger: Hello

---- PASS: TestCreateDefaultChain (0.00s)
---- PASS:
TestCreateDefaultChain/3_loggers,_2_of_them_writes_to_console,_second_only_if_it_founds_the_word_'hello',_thirdWrites_to_some_variable_if_second_found_'hello' (0.00s)
PASS
ok
```

The first half of the test prints two messages--the `First logger: message that breaks the chain`, which is the expected message for the `FirstLogger`. But it halts in the `SecondLogger` because no `hello` word has been found on the incoming message; that's why it prints the `Finishing in second logging` string.

The second half of the test receives the message `Hello`. So the `FirstLogger` prints and the `SecondLogger` prints too. The third logger doesn't print to console at all but to our `myWriter.receivedMessage` line defined in the test.

What about a closure?

Sometimes it can be useful to define an even more flexible link in the chain for quick debugging. We can use closures for this so that the link functionality is defined by the caller. What does a closure link look like? Similar to the `WriterLogger` logger:

```
type ClosureChain struct {
    NextChain ChainLogger
    Closure   func(string)
}

func (c *ClosureChain) Next(s string) {
    if c.Closure != nil {
        c.Closure(s)
    }

    if c.NextChain != nil {
        c.Next(s)
    }
}
```

The `ClosureChain` type has a `NextChain`, as usual, and a `Closure` member. Look at the signature of the `Closure: func(string)`. This means it is a function that takes a `string` and returns nothing.

The `Next(string)` method for `ClosureChain` checks that the `Closure` member is stored and executes it with the incoming string. As usual, the link checks for more links to pass the message as every link in the chain.

So, how do we use it now? We'll define a new test to show its functionality:

```
t.Run("2 loggers, second uses the closure implementation", func(t
*testing.T) {
    myWriter = myTestWriter{}
    closureLogger := ClosureChain{
        Closure: func(s string) {
            fmt.Printf("My closure logger! Message: %s\n", s)
            myWriter.receivedMessage = &s
        },
    }

    writerLogger.NextChain = &closureLogger

    chain.Next("Hello closure logger")

    if *myWriter.receivedMessage != "Hello closure logger" {
        t.Fatal("Expected message wasn't received in myWriter")
    }
})
```

```
}
```

The description of this test makes it clear: "2 loggers, second uses the closure implementation". We simply use two `ChainLogger` implementations and we use the `closureLogger` in the second link. We have created a new `myTestWriter` to store the contents of the message. When defining the `ClosureChain`, we defined an anonymous function directly on the `Closure` member when creating `closureLogger`. It prints "My closure logger! Message: %s\n" with the incoming message replacing "%s". Then, we store the incoming message on `myWriter`, to check later.

After defining this new link, we use the third link from the previous test, add the closure as the fourth link, and passed the message Hello closure logger. We use the word Hello at the beginning so that we ensure that the message will pass the `SecondLogger`.

Finally, the contents of `myWriter.receivedMessage` must contain the pased text: Hello closure logger. This is quite a flexible approach with one drawback: when defining a closure like this, we cannot test its contents in a very elegant way. Let's run the tests again:

```
go test -v .
==== RUN    TestCreateDefaultChain
==== RUN
TestCreateDefaultChain/3_loggers,_2_of_themWritesToConsole,_secondOnlyIfItFoundsTheWord'hello',_thirdWritesToSomeVariableIfSecondFound'hello'
First logger: message that breaks the chain
Finishing in second logging

First logger: Hello
Second logger: Hello
==== RUN
TestCreateDefaultChain/2_loggers,_secondUsesTheClosureImplementation
First logger: Hello closure logger
Second logger: Hello closure logger
My closure logger! Message: Hello closure logger
--- PASS: TestCreateDefaultChain (0.00s)
    --- PASS:
TestCreateDefaultChain/3_loggers,_2_of_themWritesToConsole,_secondOnlyIfItFoundsTheWord'hello',_thirdWritesToSomeVariableIfSecondFound'hello' (0.00s)
    --- PASS:
TestCreateDefaultChain/2_loggers,_secondUsesTheClosureImplementation (0.00s)
PASS
ok
```

Look at the third RUN: the message passes correctly through the first, second, and third links to arrive at the closure that prints the expected My closure logger! Message: Hello closure logger message.

It's very useful to add a closure method implementation to some interfaces as it provides quite a lot of flexibility when using the library. You can find this approach very often in Go code, being the most known the one of package `net/http`. The `HandleFunc` function which we used previously in the structural patterns to define a handler for an HTTP request.

Putting it together

We learned a powerful tool to achieve dynamic processing of actions and state handling. The Chain of responsibility pattern is widely used, also to create **Finite State Machines (FSM)**. It is also used interchangeably with the Decorator pattern with the difference that when you decorate, you change the structure of an object while with the chain you define a behavior for each link in the chain that can break it too.

Command design pattern

To finish with this chapter, we will see also the **Command** pattern--a tiny design pattern but still frequently used. You need a way to connect types that are really unrelated? So design a Command for them.

Description

The Command design pattern is quite similar to the Strategy design pattern but with key differences. While in the strategy pattern we focus on changing algorithms, in the Command pattern, we focus on the invocation of something or on the abstraction of some type.

A Command pattern is commonly seen as a container. You put something like the info for user interaction on a UI that could be `click on login` and pass it as a command. You don't need to have the complexity related to the `click on login` action in the command but simply the action itself.

An example for the organic world would be a box for a delivery company. We can put anything on it but, as a delivery company, we are interested in managing the box instead of its contents directly.

The command pattern will be used heavily when dealing with channels. With channels you can send any message through it but, if we need a response from the receiving side of the channel, a common approach is to create a command that has a second, response channel attached where we are listening.

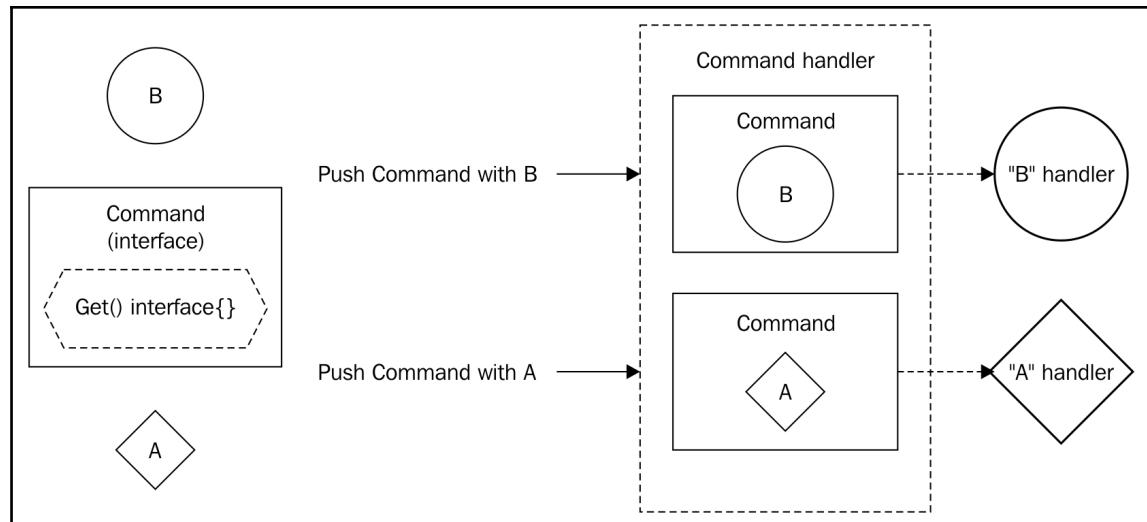
Similarly, a good example would be a multi-player video game, where every stroke of each user can be sent as commands to the rest of the users through the network.

Objectives

When using the Command design pattern, we are trying to encapsulate some sort of action or information in a light package that must be processed somewhere else. It's similar to the Strategy pattern but, in fact, a Command could trigger a preconfigured Strategy somewhere else, so they are not the same. The following are the objectives for this design pattern:

- Put some information into a box. Just the receiver will open the box and know its contents.
- Delegate some action somewhere else.

The behavior is also explained in the following diagram:



There we have a **Command** interface with a **Get()** method. We have a type **A** and a type **B**. The idea is that **A** and **B** implement the **Command** interface to return themselves as an `interface{}`. As now they implement **Command**, they can be used in a **Command handler** which doesn't care very much about the underlying type. Now **A** and **B** can travel through functions that handles commands or store Commands freely. But **B** handler can take an object from any **Command** handler to "unwrap" it and take its **B** content as well as **A** command handler with its **A** content.

We put the information in a box (the **Command**) and delegate what to do with it to the handlers of Commands.

A simple queue

Our first example is going to be pretty small. We will put some information into a Command implementer and we will have a queue. We will create many instances of a type implementing a Command pattern and we will pass them to a queue that will store the commands until three of them are in the queue, at which time it will process them.

Acceptance criteria

So the ideal acceptance criteria to understand well the implications of the Command should reflect somehow the creation of a box that can accept unrelated types and the execution of the Command itself:

- We need a constructor of console printing commands. When using this constructor with a `string`, it will return a command that will print it. In this case, the handler is inside the command that acts as a box and as a handler.
- We need a data structure that stores incoming commands in a queue and prints them once the queue reaches the length of three.

Implementation

This pattern is quite simple and we will write a few different examples so we'll implement the library directly to keep things light and short. The classical Command design pattern usually has a common type structure with an `Execute` method. We are also going to use this structure as it's quite flexible and simple:

```
type Command interface {
    Execute()
}
```

This is generic enough to fill a lot of unrelated types! Think about it--we are going to create a type that prints to console when using the `Execute()` method but it could print a number or launch a rocket as well! The key here is to focus on invocations because the handlers are also in Command. So we need some type implementing this interface and printing to the console some sort of message:

```
type ConsoleOutput struct {
    message string
}

func (c *ConsoleOutput) Execute() {
    fmt.Println(c.message)
}
```

The `ConsoleOutput` type implements the `Command` interface and prints to the console the member called `message`.

As defined in the first acceptance criterion, we need a `Command` constructor that accepts a message string and returns the `Command` interface. It will have the signature `func CreateCommand(s string) Command`:

```
func CreateCommand(s string) Command {
    fmt.Println("Creating command")

    return &ConsoleOutput{
        message: s,
    }
}
```

For the command queue, we'll define a very simple type called `CommandQueue` to store in a queue any type implementing the `Command` interface:

```
type CommandQueue struct {
    queue []Command
}

func (p *CommandQueue) AddCommand(c Command) {
    p.queue = append(p.queue, c)

    if len(p.queue) == 3 {
        for _, command := range p.queue {
            command.Execute()
        }

        p.queue = make([]Command, 3)
    }
}
```

The `CommandQueue` type stores an array of the `Commands` interface. When the queue array reaches three items, it executes all the commands stored in the `queue` field. If it hasn't reached the required length yet, it just stores the command.

We will create five commands, enough to trigger the command queue mechanism, and add them to the queue. Each time a command is created, the message `Creating` command will be printed to the console. When we create the third command, the automatic command executor will be launched, printing the first three messages. We create and add two commands more, but because we haven't reached the third command again, they won't be printed and just the `Creating` command messages will be printed:

```
func main() {
    queue := CommandQueue{}

    queue.AddCommand(CreateCommand("First message"))
    queue.AddCommand(CreateCommand("Second message"))
    queue.AddCommand(CreateCommand("Third message"))

    queue.AddCommand(CreateCommand("Fourth message"))
    queue.AddCommand(CreateCommand("Fifth message"))
}
```

Let's run the `main` program. Our definition said that the commands are processed once every three messages and we will create a total of five messages. The first three messages must be printed but not the fourth and fifth because we didn't reach a sixth message to trigger the command processing:

```
$go run command.go

Creating command
Creating command
Creating command
First message
Second message
Third message
Creating command
Creating command
```

As you can see, the fourth and fifth messages aren't printed, as expected, but we know that the commands were created and stored on the array. They just weren't processed because the queue was waiting for one command more to trigger the processor.

More examples

The previous example shows how to use a Command handler that executes the content of the command. But a common way to use a Command pattern is to delegate the information, instead of the execution, to a different object.

For example, instead of printing to the console, we will create a command that extracts information:

```
type Command interface {
    Info() string
}
```

In this case, our `Command` interface will have a method named `Info` that will retrieve some information from its implementor. We will create two implementations; one will return the time passed since the creation of the command to its execution:

```
type TimePassed struct {
    start time.Time
}

func (t *TimePassed) Info() string {
    return time.Since(t.start).String()
}
```

The `time.Since` function returns the time elapsed since the time stored in the provided parameter. We returned the string representation of the passed time by calling the `String()` method on the `time.Time` type. The second implementation of our new Command will return the message Hello World!:

```
type HelloMessage struct{ }

func (h HelloMessage) Info() string {
    return "Hello world!"
}
```

And our `main` function will simply create an instance of each type, then waits for a second and print the info returned from each Command:

```
func main() {
    var timeCommand Command
    timeCommand = &TimePassed{time.Now()}

    var helloCommand Command
    helloCommand = &HelloMessage{}

    time.Sleep(time.Second)

    fmt.Println(timeCommand.Info())
    fmt.Println(helloCommand.Info())
}
```

The `time.Sleep` function stops the execution of the current goroutine for the specified period (a second). So, to recall--the `timeCommand` variable stores the time when the program was started and its `Info()` method returns a string representation of the time that passed since we give a value to the type to the moment were we called the `Info()` method on it. The `helloCommand` variable returns the message Hello World! when we call its `Info()` method. Here we haven't implemented a Command handler again to keep things simple but we can consider the console as the handler because we can only print ASCII characters on it like the ones retrieved by the `Info()` method.

Let's run the `main` function:

```
go run command.go
1.000216755s
Hello world!
```

Here we are. In this case, we retrieve some information by using the Command pattern. One type stores time information while the other stores nothing and it simply returns the same simple string. Each time we run the main function will return a different elapsed time, so don't worry if the time doesn't match with the one in the example.

Chain of responsibility of commands

Do you remember the chain of responsibility design pattern? We were passing a string message between links to print its contents. But we could be using the previous Command to retrieve information for logging to the console. We'll mainly reuse the code that we have written already.

The Command interface will be from the type interface that returns a string from the previous example:

```
type Command interface {
    Info() string
}
```

We will use the Command implementation of the TimePassed type too:

```
type TimePassed struct {
    start time.Time
}

func (t *TimePassed) Info() string {
    return time.Since(t.start).String()
}
```

Remember that this type returns the elapsed time from the object creation on its Info() string method. We also need the ChainLogger interface from the *Chain of responsibility design pattern* section of this chapter but, this time, it will pass Commands on its Next method instead of string:

```
type ChainLogger interface {
    Next(Command)
}
```

We'll use just the same type for two links in the chain for simplicity. This link is very similar to the `FirstLogger` type from the chain of responsibility example, but this time it will append the message `Elapsed time from creation:` and it will wait 1 second before printing. We'll call it `Logger` instead of `FirstLogger`:

```
type Logger struct {
    NextChain ChainLogger
}

func (f *Logger) Next(c Command) {
    time.Sleep(time.Second)

    fmt.Printf("Elapsed time from creation: %s\n", c.Info() )

    if f.NextChain != nil {
        f.NextChain.Next(c)
    }
}
```

Finally, we need a `main` function to execute the chain that takes `Command` pointers:

```
func main() {
    second := new(Logger)
    first := Logger{NextChain: second}

    command := &TimePassed{start: time.Now()}

    first.Next(command)
}
```

Line by line, we create a variable called `second` with a pointer to a `Logger`; this is going to be the second link in our chain. Then we create a variable called `first`, that will be the first link in the chain. The first link points to the `second` variable, the second link in the chain.

Then, we create an instance of `TimePassed` to use it as the `Command` type. The start time of this command is the execution time (the `time.Now()` method returns the time in the moment of the execution).

Finally, we pass the `Command` interface to the chain on the `first.Next(command)` statement. The output of this program is the following:

```
go run chain_command.go
Elapsed time from creation: 1.0003419s
Elapsed time from creation: 2.000682s
```

The resulting output is reflected in the following diagram: The command with the time field is pushed to the first link that knows how to execute Commands of any type. Then it passes the Command to the second link that also knows how to execute Commands:

This approach hides the complexity behind each Command execution from the Command handlers on each link. The functionality hidden behind a Command can be simple or incredibly complex but the idea here is to reuse the handler to manage many types of unrelated implementations.

Rounding-up the Command pattern up

Command is a very tiny design pattern; its functionality is quite easy to understand but it's widely used for its simplicity. It looks very similar to the Strategy pattern but remember that Strategy is about having many algorithms to achieve some specific task, but all of them achieve the same task. In the Command pattern, you have many tasks to execute, and not all of them need to be equal.

So, in short, the Command pattern is about execution encapsulation and delegation so that just the receiver or receivers trigger that execution.

Summary

We have taken our first steps in the Behavioral patterns. The objective of this chapter was to introduce the reader to the concept of algorithm and execution encapsulation using proper interfaces and structures. With the strategy, we have encapsulated algorithms, with the chain of responsibility handlers and with the Command design pattern executions.

Now, with the knowledge we have acquired about the strategy pattern, we can uncouple heavily our applications from their algorithms, just for testing, this is a very useful feature to inject mocks in different types that would be almost impossible to test. But also for anything that could need different approaches based on some context (such as shorting a list; some algorithms perform better depending on the distribution of the list).

The Chain of Responsibility pattern opens the door of middleware of any type and plugin-like libraries to improve the functionality of some part. Many open source projects uses a Chain of Responsibility to handler HTTP requests and responses to extract information to the end user (such as cookies info) or check authentication details (I'll let you pass to the next link only if I have you on my database).

Finally, the Command pattern is the most common pattern for UI handling but also very useful in many other scenarios where we need some type of handling between many unrelated types that are travelling through the code (such as a message passed through a channel).

6

Behavioral Patterns - Template, Memento, and Interpreter Design Patterns

In this chapter, we will see the next three Behavioral design patterns. The difficulty is being raised as now we will use combinations of Structural and Creational patterns to better solve the objective of some of the Behavioral patterns.

We will start with Template design pattern, a pattern that looks very similar to the Strategy pattern but that provides greater flexibility. Memento design pattern is used in 99% of applications we use every day to achieve undo functions and transactional operations. Finally, we will write a reverse polish notation interpreter to perform simple mathematical operations.

Let's start with the Template design pattern.

Template design pattern

The **Template** pattern is one of those widely used patterns that are incredibly useful, especially when writing libraries and frameworks. The idea is to provide a user some way to execute code within an algorithm.

In this section, we will see how to write idiomatic Go Template patterns and see some Go source code where it's wisely used. We will write an algorithm of three steps where the second step is delegated to the user while the first and third aren't. The first and third steps on the algorithm represent the template.

Description

While with the Strategy pattern we were encapsulating algorithm implementation in different strategies, with the Template pattern we will try to achieve something similar but with just part of the algorithm.

The Template design pattern lets the user write a part of an algorithm while the rest is executed by the abstraction. This is common when creating libraries to ease in some complex task or when reusability of some algorithm is compromised by only a part of it.

Imagine, for example, that we have a long transaction of HTTP requests. We have to perform the following steps:

1. Authenticate user.
2. Authorize him.
3. Retrieve some details from a database.
4. Make some modification.
5. Send the details back in a new request.

It wouldn't make sense to repeat steps 1 to 5 in the user's code every time he needs to modify something on the database. Instead, steps 1, 2, 3, and 5 will be abstracted in the same algorithm that receives an interface with whatever the fifth step needs to finish the transaction. It doesn't need to be a interface either, it could be a callback.

Objectives

The Template design pattern is all about reusability and giving responsibilities to the user. So the objectives for this pattern are following:

- Defer a part of an algorithm of the library to a the user
- Improve reusability by abstracting the parts of the code that are not common between executions

Example - a simple algorithm with a deferred step

In our first example, we are going to write an algorithm that is composed of three steps and each of them returns a message. The first and third steps are controlled by the Template and just the second step is deferred to the user.

Requirements and acceptance criteria

A brief description of what the Template pattern has to do is to define a template for an algorithm of three steps that defers the implementation of the second step to the user:

1. Each step in the algorithm must return a string.
2. The first step is a method called `first()` and returns the string `hello`.
3. The third step is a method called `third()` and returns the string `template`.
4. The second step is whatever string the user wants to return but it's defined by the `MessageRetriever` interface that has a `Message()` string method.
5. The algorithm is executed sequentially by a method called `ExecuteAlgorithm` and returns the strings returned by each step joined in a single string by a space.

Unit tests for the simple algorithm

We will focus on testing the public methods only. This is a very common approach. All in all, if your private methods aren't called from some level of the public ones, they aren't called at all. We need two interfaces here, one for the Template implementors and one for the abstract step of the algorithm:

```
type MessageRetriever interface {
    Message() string
}

type Template interface {
    first() string
    third() string
    ExecuteAlgorithm(MessageRetriever) string
}
```

A Template implementor will accept a `MessageRetriever` interface to execute as part of its execution algorithm. We need a type that implements this interface called `Template`, we will call it `TemplateImpl`:

```
type TemplateImpl struct{ }

func (t *TemplateImpl) first() string {
    return ""
}

func (t *TemplateImpl) third() string {
    return ""
}
```

```
func (t *TemplateImpl) ExecuteAlgorithm(m MessageRetriever) string {
    return ""
}
```

So our first test checks the fourth and fifth acceptance criteria. We will create the `TestStruct` type that implements the `MessageRetriever` interface returning the string `world` and has embedded the `Template` so that it can call the `ExecuteAlgorithm` method. It will act as the `Template` and the abstraction:

```
type TestStruct struct {
    Template
}

func (m *TestStruct) Message() string {
    return "world"
}
```

First, we will define the `TestStruct` type. In this case, the part of the algorithm deferred to us is going to return the `world` text. This is the string we will look for later in the test doing a check of type "is the word `world` present on this string?".

Take a close look, the `TestStruct` embeds a type called `Template` which represents the `Template` pattern of our algorithm.

When we implement the `Message()` method, we are implicitly implementing the `MessageRetriever` interface. So now we can use `TestStruct` type as a pointer to a `MessageRetriever` interface:

```
func TestTemplate_ExecuteAlgorithm(t *testing.T) {
    t.Run("Using interfaces", func(t *testing.T){
        s := &TestStruct{}
        res := s.ExecuteAlgorithm(s)
        expected := "world"

        if !strings.Contains(res, expected) {
            t.Errorf("Expected string '%s' wasn't found on returned string\n",
                    expected)
        }
    })
}
```

In the test, we will use the type we have just created. When we call the `ExecuteAlgorithm` method, we need to pass the `MessageRetriever` interface. As the `TestStruct` type also implements the `MessageRetriever` interface, we can pass it as an argument, but this is not mandatory, of course.

The result of the `ExecuteAlgorithm` method, as defined in the fifth acceptance criterion, must return a string that contains the returned value of the `first()` method, the returned value of `TestStruct` (the `world` string) and the returned value of the `third()` method separated by a space. Our implementation is on the second place; that's why we checked that a space is prefixed and suffixed on the string `world`.

So, if the returned string, when calling the `ExecuteAlgorithm` method, doesn't contain the string `world`, the test fails.

This is enough to make the project compile and run the tests that should fail:

```
go test -v .
==== RUN    TestTemplate_ExecuteAlgorithm
==== RUN    TestTemplate_ExecuteAlgorithm/Using_interfaces
--- FAIL: TestTemplate_ExecuteAlgorithm (0.00s)
    --- FAIL: TestTemplate_ExecuteAlgorithm/Using_interfaces (0.00s)
        template_test.go:47: Expected string ' world ' was not found on
        returned string
FAIL
exit status 1
FAIL
```

Time to pass to the implementation of this pattern.

Implementing the Template pattern

As defined in the acceptance criteria, we have to return the string `hello` in the `first()` method and the string `template` in the `third()` method. That's pretty easy to implement:

```
type Template struct{}

func (t *Template) first() string {
    return "hello"
}

func (t *Template) third() string {
    return "template"
}
```

With this implementation, we should be covering the `second` and `third` acceptance criteria and partially covering the `first` criterion (each step in the algorithm must return a string).

To cover the *fifth* acceptance criterion, we define an `ExecuteAlgorithm` method that accepts the `MessageRetriever` interface as argument and returns the full algorithm: a single string done by joining the strings returned by the `first()`, `Message()` string and `third()` methods:

```
func (t *Template) ExecuteAlgorithm(m MessageRetriever) string {
    return strings.Join([]string{t.first(), m.Message(), t.third()}, " ")}
```

The `strings.Join` function has the following signature:

```
func Join([]string, string) string
```

It takes an array of strings and joins them, placing the second argument between each item in the array. In our case, we create a string array on the fly to pass it as the first argument. Then we pass a whitespace as the second argument.

With this implementation, the tests must be passing already:

```
go test -v .
==== RUN    TestTemplate_ExecuteAlgorithm
==== RUN    TestTemplate_ExecuteAlgorithm/Using_interfaces
--- PASS: TestTemplate_ExecuteAlgorithm (0.00s)
    --- PASS: TestTemplate_ExecuteAlgorithm/Using_interfaces (0.00s)
PASS
ok
```

The tests passed. The test has checked that the string `world` is present in the returned result, which is the `hello world` template message. The `hello` text was the string returned by the `first()` method, the `world` string was returned by our `MessageRetriever` implementation, and `template` was the string returned by the `third()` method. The whitespaces are inserted by Go's `strings.Join` function. But any use of the `TemplateImpl.ExecuteAlgorithm` type will always return "hello [something] template" in its result.

Anonymous functions

This is not the only way to achieve the Template design pattern. We can also use an anonymous function to give our implementation to the `ExecuteAlgorithm` method.

Let's write a test in the same method that was used previously just after the test (marked in bold):

```
func TestTemplate_ExecuteAlgorithm(t *testing.T) {
    t.Run("Using interfaces", func(t *testing.T){
        s := &TestStruct{}
        res := s.ExecuteAlgorithm(s)

        expectedOrError(res, " world ", t)
    })

    t.Run("Using anonymous functions", func(t *testing.T){
        m := new(AnonymousTemplate)
        res := m.ExecuteAlgorithm(func() string {
            return "world"
        })
        expectedOrError(res, " world ", t)
    })
}

func expectedOrError(res string, expected string, t *testing.T){
    if !strings.Contains(res, expected) {
        t.Errorf("Expected string '%s' was not found on returned string\n",
            expected)
    }
}
```

Our new test is called *Using anonymous functions*. We have also extracted the checking on the test to an external function to reuse it in this test. We have called this function `expectedOrError` because it will fail with an error if the expected value isn't received.

In our test, we will create a type called `AnonymousTemplate` that replaces the previous `Template` type. The `ExecuteAlgorithm` method of this new type accepts the `func()` method `string` type that we can implement directly in the test to return the string `world`.

The `AnonymousTemplate` type will have the following structure:

```
type AnonymousTemplate struct{}

func (a *AnonymousTemplate) first() string {
    return ""
}

func (a *AnonymousTemplate) third() string {
    return ""
}
```

```
func (a *AnonymousTemplate) ExecuteAlgorithm(f func() string) string {
    return ""
}
```

The only difference with the `Template` type is that the `ExecuteAlgorithm` method accepts a function that returns a string instead of a `MessageRetriever` interface. Let's run the new test:

```
go test -v .
==== RUN  TestTemplate_ExecuteAlgorithm
==== RUN  TestTemplate_ExecuteAlgorithm/Using_interfaces
==== RUN  TestTemplate_ExecuteAlgorithm/Using_anonymous_functions
--- FAIL: TestTemplate_ExecuteAlgorithm (0.00s)
    --- PASS: TestTemplate_ExecuteAlgorithm/Using_interfaces (0.00s)
    --- FAIL: TestTemplate_ExecuteAlgorithm/Using_anonymous_functions
(0.00s)
    template_test.go:47: Expected string ' world ' was not found on
returned string
FAIL
exit status 1
FAIL
```

As you can read in the output of the test execution, the error is thrown on the *Using anonymous functions* test, which is what we were expecting. Now we will write the implementation as follows:

```
type AnonymousTemplate struct{ }

func (a *AnonymousTemplate) first() string {
    return "hello"
}

func (a *AnonymousTemplate) third() string {
    return "template"
}

func (a *AnonymousTemplate) ExecuteAlgorithm(f func() string) string {
    return strings.Join([]string{a.first(), f(), a.third()}, " ")
}
```

The implementation is quite similar to the one in the `Template` type. However, now we have passed a function called `f` that we will use as the second item in the string array we used on `Join` function. As `f` is simply a function that returns a string, the only thing we need to do with it is to execute it in the proper place (the second position in the array).

Run the tests again:

```
go test -v .
==== RUN    TestTemplate_ExecuteAlgorithm
==== RUN    TestTemplate_ExecuteAlgorithm/Using_interfaces
==== RUN    TestTemplate_ExecuteAlgorithm/Using_anonymous_functions
--- PASS: TestTemplate_ExecuteAlgorithm (0.00s)
    --- PASS: TestTemplate_ExecuteAlgorithm/Using_interfaces (0.00s)
    --- PASS: TestTemplate_ExecuteAlgorithm/Using_anonymous_functions
(0.00s)
PASS
ok
```

Awesome! Now we know two ways to implement the Template design pattern.

How to avoid modifications on the interface

The problem of the previous approach is that now we have two templates to maintain and we could end duplicating code. What can we do in the situation that we cannot change the interface are we using? Our interface was `MessageRetriever` but we want to use an anonymous function now.

Well, do you remember the Adapter design pattern? We just have to create an `Adapter` type that, accepting a `func() string` type, returns an implementation of the `MessageRetriever` interface. We will call this type `TemplateAdapter`:

```
type TemplateAdapter struct {
    myFunc func() string
}

func (a *TemplateAdapter) Message() string {
    return ""
}

func MessageRetrieverAdapter(f func() string) MessageRetriever {
    return nil
}
```

As you can see, the `TemplateAdapter` type has a field called `myFunc` which is of type `func() string`. We have also defined `adapter` as private because it shouldn't be used without a function defined in the `myFunc` field. We have created a public function called the `MessageRetrieverAdapter` to achieve this. Our test should look more or less like this:

```
t.Run("Using anonymous functions adapted to an interface", func(t
*testing.T){
```

```
messageRetriever := MessageRetrieverAdapter(func() string {
    return "world"
})

if messageRetriever == nil {
    t.Fatal("Can not continue with a nil MessageRetriever")
}

template := Template{}
res := template.ExecuteAlgorithm(messageRetriever)

expectedOrError(res, " world ", t)
})
```

Look at the statement where we called the `MessageRetrieverAdapter` method. We passed an anonymous function as an argument defined as `func() string`. Then, we reuse the previously defined `Template` type from our first test to pass the `messageRetriever` variable. Finally, we checked again with the `expectedOrError` method. Take a look at the `MessageRetrieverAdapter` method, it will return a function that has `nil` value. If strictly following the test-driven development rules, we must do tests first and they must not pass before implementation is done. That's why we returned `nil` on the `MessageRetrieverAdapter` function.

So, let's run the tests:

```
go test -v .
==== RUN TestTemplate_ExecuteAlgorithm
==== RUN TestTemplate_ExecuteAlgorithm/Using_interfaces
==== RUN TestTemplate_ExecuteAlgorithm/Using_anonymous_functions
==== RUN TestTemplate_ExecuteAlgorithm/Using_anonymous_functions_adapted_to_an_interface
---- FAIL: TestTemplate_ExecuteAlgorithm (0.00s)
    ---- PASS: TestTemplate_ExecuteAlgorithm/Using_interfaces (0.00s)
    ---- PASS: TestTemplate_ExecuteAlgorithm/Using_anonymous_functions (0.00s)
    ---- FAIL:
TestTemplate_ExecuteAlgorithm/Using_anonymous_functions_adapted_to_an_interface (0.00s)
    template_test.go:39: Can not continue with a nil MessageRetriever
FAIL
exit status 1
FAIL
```

The test fails on *line 39* of the code and it doesn't continue (again, depending on how you wrote your code, the line representing your error could be somewhere else). We stop test execution because we will need a valid `MessageRetriever` interface when we call the `ExecuteAlgorithm` method.

For the implementation of the adapter for our Template pattern, we will start with `MessageRetrieverAdapter` method:

```
func MessageRetrieverAdapter(f func() string) MessageRetriever {
    return &adapter{myFunc: f}
}
```

It's very easy, right? You could be wondering what happens if we pass `nil` value for the `f` argument. Well, we will cover this issue by calling the `myFunc` function.

The adapter type is finished with this implementation:

```
type adapter struct {
    myFunc func() string
}

func (a *adapter) Message() string {
    if a.myFunc != nil {
        return a.myFunc()
    }

    return ""
}
```

When calling the `Message()` function, we check that we actually have something stored in the `myFunc` function before calling. If nothing was stored, we return an empty string.

Now, our third implementation of the `Template` type, using the Adapter pattern, is done:

```
go test -v .
==== RUN    TestTemplate_ExecuteAlgorithm
==== RUN    TestTemplate_ExecuteAlgorithm/Using_interfaces
==== RUN    TestTemplate_ExecuteAlgorithm/Using_anonymous_functions
==== RUN
TestTemplate_ExecuteAlgorithm/Using_anonymous_functions_adapted_to_an_interface
--- PASS: TestTemplate_ExecuteAlgorithm (0.00s)
    --- PASS: TestTemplate_ExecuteAlgorithm/Using_interfaces (0.00s)
    --- PASS: TestTemplate_ExecuteAlgorithm/Using_anonymous_functions
(0.00s)
    --- PASS:
TestTemplate_ExecuteAlgorithm/Using_anonymous_functions_adapted_to_an_interface
```

```
face (0.00s)
PASS
ok
```

Looking for the Template pattern in Go's source code

The `Sort` package in Go's source code can be considered a Template implementation of a sort algorithm. As defined in the package itself, the `Sort` package provides primitives for sorting slices and user-defined collections.

Here, we can also find a good example of why Go authors aren't worried about implementing generics. Sorting the lists is maybe the best example of generic usage in other languages. The way that Go deals with this is very elegant too—it deals with this issue with an interface:

```
type Interface interface {
    Len() int
    Less(i, j int) bool
    Swap(i, j int)
}
```

This is the interface for lists that need to be sorted by using the `sort` package. In the words of Go's authors:

"A type, typically, is a collection that satisfies sort. Interface can be sorted by the routines in this package. The methods require that the elements of the collection be enumerated by an integer index."

In other words, write a type that implements this `Interface` so that the `Sort` package can be used to sort any slice. The sorting algorithm is the template and we must define how to retrieve values in our slice.

If we peek in the `sort` package, we can also find an example of how to use the sorting template but we will create our own example:

```
package main

import (
    "sort"
    "fmt"
)

type MyList []int
```

```
func (m MyList) Len() int {
    return len(m)
}

func (m MyList) Swap(i, j int) {
    m[i], m[j] = m[j], m[i]
}

func (m MyList) Less(i, j int) bool {
    return m[i] < m[j]
}
```

First, we have done a very simple type that stores an `int` list. This could be any kind of list, usually a list of some kind of struct. Then we have implemented the `sort.Interface` interface by defining the `Len`, `Swap`, and `Less` methods.

Finally, the `main` function creates an unordered list of numbers of the `MyList` type:

```
func main() {
    var myList MyList = []int{6, 4, 2, 8, 1}

    fmt.Println(myList)
    sort.Sort(myList)
    fmt.Println(myList)
}
```

We print the list that we created (unordered) and then we sort it (the `sort.Sort` method actually modifies our variable instead of returning a new list so beware!). Finally, we print again the resulting list. The console output of this `main` method is the following:

```
go run sort_example.go
[6 4 2 8 1]
[1 2 4 6 8]
```

The `sort.Sort` function has sorted our list in a transparent way. It has a lot of code written and delegates `Len`, `Swap` and `Less` methods to an interface, like we did in our template delegating to the `MessageRetriever` interface.

Summarizing the Template design pattern

We wanted to put a lot of focus on this pattern because it is very important when developing libraries and frameworks and allows a lot of flexibility and control to users of our library.

We have also seen again that it's very common to mix patterns to provide flexibility to the users, not only in a behavioral way but also structural. This will come very handy when working with concurrent apps where we need to restrict access to parts of our code to avoid races.

Memento design pattern

Let's now look at a pattern with a fancy name. If we check a dictionary to see the meaning of *memento*, we will find the following description:

"An object kept as a reminder of a person or event."

Here, the key word is **reminder** as we will remember actions with this design pattern.

Description

The meaning of memento is very similar to the functionality it provides in design patterns. Basically, we'll have a type with some state and we want to be able to save milestones of its state. Having a finite amount of states saved, we can recover them if necessary for a variety of tasks-undo operations, historic, and so on.

The Memento design pattern usually has three players (usually called **actors**):

- **Memento:** A type that stores the type we want to save. Usually, we won't store the business type directly and we provide an extra layer of abstraction through this type.
- **Originator:** A type that is in charge of creating mementos and storing the current active state. We said that the Memento type wraps states of the business type and we use originator as the creator of mementos.
- **Care Taker:** A type that stores the list of mementos that can have the logic to store them in a database or to not store more than a specified number of them.

Objectives

Memento is all about a sequence of actions over time, say to undo one or two operations or to provide some kind of transactionality to some application.

Memento provides the foundations for many tasks, but its main objectives could be defined as follows:

- Capture an object state without modifying the object itself
- Save a limited amount of states so we can retrieve them later

A simple example with strings

We will develop a simple example using a string as the state we want to save. This way, we will focus on the common Memento pattern implementations before making it a bit more complex with a new example.

The string, stored in a field of a `State` instance, will be modified and we will be able to undo the operations done in this state.

Requirements and acceptance criteria

We are constantly talking about state; all in all, the Memento pattern is about storing and retrieving states. Our acceptance criteria must be all about states:

1. We need to store a finite amount of states of type string.
2. We need a way to restore the current stored state to one of the state list.

With these two simple requirements, we can already start writing some tests for this example.

Unit test

As mentioned previously, the Memento design pattern is usually composed of three actors: state, memento, and originator. So we will need three types to represent these actors:

```
type State struct {
    Description string
}
```

The `State` type is the core business object we will be using during this example. It's any kind of object that we want to track:

```
type memento struct {
    state State
}
```

The `memento` type has a field called `state` representing a single value of a `State` type. Our states will be containerized within this type before storing them into the `care_taker` type. You could be wondering why we don't store directly `State` instances. Basically, because it will couple the `originator` and the `careTaker` to the business object and we want to have as little coupling as possible. It will also be less flexible, as we will see in the second example:

```
type originator struct {
    state State
}

func (o *originator) NewMemento() memento {
    return memento{}
}

func (o *originator) ExtractAndStoreState(m memento) {
    //Does nothing
}
```

The `originator` type also stores a state. The `originator` struct's objects will take states from mementos and create new mementos with their stored state.



What's the difference between the `originator` object and the `Memento` pattern? Why don't we use `Originator` pattern's object directly? Well, if the `Memento` contains a specific state, the `originator` type contains the state that is currently loaded. Also, to save the state of something could be as simple as to take some value or as complex as to maintain the state of some distributed application.

The `Originator` will have two public methods--the `NewMemento()` method and the `ExtractAndStoreState(m memento)` method. The `NewMemento` method will return a new `Memento` built with `originator` current `State` value. The `ExtractAndStoreState` method will take the state of a `Memento` and store it in the `Originator`'s `state` field:

```
type careTaker struct {
    mementoList []memento
}
```

```
func (c *careTaker) Add(m memento) {
    //Does nothing
}

func (c *careTaker) Memento(i int) (memento, error) {
    return memento{}, fmt.Errorf("Not implemented yet")
}
```

The `careTaker` type stores the `Memento` list with all the states we need to save. It also stores an `Add` method to insert a new `Memento` on the list and a `Memento` retriever that takes an index on the `Memento` list.

So let's start with the `Add` method of the `careTaker` type. The `Add` method must take a `memento` object and add it to the `careTaker` object's list of `Mementos`:

```
func TestCareTaker_Add(t *testing.T) {
    originator := originator{}
    originator.state = State{Description:"Idle"}

    careTaker := careTaker{}
    mem := originator.NewMemento()
    if mem.state.Description != "Idle" {
        t.Error("Expected state was not found")
    }
}
```

At the beginning of our test, we created two basic actors for `memento`--the `originator` and the `careTaker`. We set a first state on the `originator` with the description `Idle`.

Then, we create the first `Memento` calling the `NewMemento` method. This should wrap the current `originator`'s state in a `memento` type. Our first check is very simple--the state description of the returned `Memento` must be like the state description we pass to the `originator`, that is, the `Idle` description.

The last step to check whether our `Memento`'s `Add` method works correctly is to see whether the `Memento` list has grown after adding one item:

```
currentLen := len(careTaker.mementoList)
careTaker.Add(mem)

if len(careTaker.mementoList) != currentLen+1 {
    t.Error("No new elements were added on the list")
}
```

We also have to test the `Memento(int) memento` method. This should take a `memento` value from the `careTaker` list. It takes the index you want to retrieve from the list so, as usual with lists, we must check that it behaves correctly against negative numbers and out of index values:

```
func TestCareTaker_Memento(t *testing.T) {
    originator := originator{}
    careTaker := careTaker{}

    originator.state = State{"Idle"}
    careTaker.Add(originator.NewMemento())
```

We have to start like we did in our previous test--creating an `originator` and `careTaker` objects and adding the first `Memento` to the `caretaker`:

```
mem, err := careTaker.Memento(0)
if err != nil {
    t.Fatal(err)
}

if mem.state.Description != "Idle" {
    t.Error("Unexpected state")
}
```

Once we have the first object on the `careTaker` object, we can ask for it using `careTaker.Memento(0)`. Index 0 on the `Memento(int)` method retrieves the first item on the slice (remember that slices start with 0). No error should be returned because we have already added a value to the `caretaker` object.

Then, after retrieving the first `memento`, we checked that the description matches the one that we passed at the beginning of the test:

```
mem, err = careTaker.Memento(-1)
if err == nil {
    t.Fatal("An error is expected when asking for a negative number but no
error was found")
}
}
```

The last step on this test involves using a negative number to retrieve some value. In this case, an error must be returned that shows that no negative numbers can be used. It is also possible to return the first index when you pass negative numbers but here we will return an error.

The last function to check is the `ExtractAndStoreState` method. This function must take a `Memento` and extract all its state information to set it in the `Originator` object:

```
func TestOriginator_ExtractAndStoreState(t *testing.T) {
    originator := originator{state:State{"Idle"}}
    idleMemento := originator.NewMemento()

    originator.ExtractAndStoreState(idleMemento)
    if originator.state.Description != "Idle" {
        t.Error("Unexpected state found")
    }
}
```

This test is simple. We create a default `originator` variable with an `Idle` state. Then, we retrieve a new `Memento` object to use it later. We change the state of the `originator` variable to the `Working` state to ensure that the new state will be written.

Finally, we have to call the `ExtractAndStoreState` method with the `idleMemento` variable. This should restore the state of the `originator` to the `idleMemento` state's value, something that we checked in the last `if` statement.

Now it's time to run the tests:

```
go test -v .
==== RUN  TestCareTaker_Add
--- FAIL: TestCareTaker_Add (0.00s)
    memento_test.go:13: Expected state was not found
    memento_test.go:20: No new elements were added on the list
==== RUN  TestCareTaker_Memento
--- FAIL: TestCareTaker_Memento (0.00s)
    memento_test.go:33: Not implemented yet
==== RUN  TestOriginator_ExtractAndStoreState
--- FAIL: TestOriginator_ExtractAndStoreState (0.00s)
    memento_test.go:54: Unexpected state found
FAIL
exit status 1
FAIL
```

Because the three tests fail, we can continue with the implementation.

Implementing the Memento pattern

The Memento pattern's implementation is usually very simple if you don't get too crazy. The three actors (`memento`, `originator`, and `care taker`) have a very defined role in the pattern and their implementation is very straightforward:

```
type originator struct {
    state State
```

```
}

func (o *originator) NewMemento() memento {
    return memento{state: o.state}
}

func (o *originator) ExtractAndStoreState(m memento) {
    o.state = m.state
}
```

The `Originator` object needs to return a new values of `Memento` types when calling the `NewMemento` method. It also needs to store the value of a `memento` object in the `state` field of the struct as needed for the `ExtractAndStoreState` method:

```
type careTaker struct {
    mementoList []memento
}

func (c *careTaker) Push(m memento) {
    c.mementoList = append(c.mementoList, m)
}

func (c *careTaker) Memento(i int) (memento, error) {
    if len(c.mementoList) < i || i < 0 {
        return memento{}, fmt.Errorf("Index not found\n")
    }
    return c.mementoList[i], nil
}
```

The `careTaker` type is also straightforward. When we call the `Add` method, we overwrite the `mementoList` field by calling the `append` method with the value passed in the argument. This creates a new list with the new value included.

When calling the `Memento` method, we have to do a couple of checks beforehand. In this case, we check that the index is not outside of the range of the slice and that the index is not a negative number in the `if` statement, in which case we return an error. If everything goes fine, it just returns the specified `memento` object and no errors.

A note about method and function naming conventions. You could find some people that like to give slightly more descriptive names to methods such as `Memento`. An example would be to use a name such as `MementoOrError` method, clearly showing that you return two objects when calling this function or even `GetMementoOrError` method. This could be a very explicit approach for naming and it's not necessarily bad, but you won't find it very common in Go's source code.



Time to check the test results:

```
go test -v .
==== RUN    TestCareTaker_Add
---- PASS: TestCareTaker_Add (0.00s)
==== RUN    TestCareTaker_Memento
---- PASS: TestCareTaker_Memento (0.00s)
==== RUN    TestOriginator_ExtractAndStoreState
---- PASS: TestOriginator_ExtractAndStoreState (0.00s)
PASS
ok
```

That was enough to reach 100% of coverage. While this is far from being a perfect metric, at least we know that we are reaching every corner of our source code and that we haven't cheated in our tests to achieve it.

Another example using the Command and Facade patterns

The previous example is good and simple enough to understand the functionality of the Memento pattern. However, it is more commonly used in conjunction with the Command pattern and a simple Facade pattern.

The idea is to use a Command pattern to encapsulate a set of different types of states (those that implement a `Command` interface) and provide a small facade to automate the insertion in the `caretaker` object.

We are going to develop a small example of a hypothetical audio mixer. We are going to use the same Memento pattern to save two types of states: `Volume` and `Mute`. The `Volume` state is going to be a byte type and the `Mute` state a Boolean type. We will use two completely different types to show the flexibility of this approach (and its drawbacks).

As a side note, we can also ship each `Command` interface with their own serialization methods on the interface. This way, we can give the ability to the caretaker to store states in some kind of storage without really knowing what's storing.

Our `Command` interface is going to have one method to return the value of its implementer. It's very simple, every command in our audio mixer that we want to undo will have to implement this interface:

```
type Command interface {
    GetValue() interface{}
}
```

There is something interesting in this interface. The `GetValue` method returns an interface to a value. This also means that the return type of this method is... well... untyped? Not really, but it returns an interface that can be a representation of any type and we will need to typecast it later if we want to use its specific type. Now we have to define the `Volume` and `Mute` types and implement the `Command` interface:

```
type Volume byte

func (v Volume) GetValue() interface{} {
    return v
}

type Mute bool

func (m Mute) GetValue() interface{} {
    return m
}
```

They are both quite easy implementations. However, the `Mute` type will return a `bool` type on the `GetValue()` method and `Volume` will return a `byte` type.

As in the previous example, we'll need a `Memento` type that will hold a `Command`. In other words, it will store a pointer to a `Mute` or a `Volume` type:

```
type Memento struct {
    memento Command
}
```

The `originator` type works as in the previous example but uses the `Command` keyword instead of the `state` keyword:

```
type originator struct {
    Command Command
}

func (o *originator) NewMemento() Memento {
    return Memento{memento: o.Command}
}

func (o *originator) ExtractAndStoreCommand(m Memento) {
    o.Command = m.memento
}
```

And the `caretaker` object is almost the same, but this time we'll use a stack instead of a simple list and we will store a command instead of a state:

```
type careTaker struct {
```

```
mementoList []Memento
}

func (c *careTaker) Add(m Memento) {
    c.mementoList = append(c.mementoList, m)
}

func (c *careTaker) Pop() Memento {
    if len(c.mementoStack) > 0 {
        tempMemento := c.mementoStack[len(c.mementoStack)-1]
        c.mementoStack = c.mementoStack[0:len(c.mementoStack)-1]
        return tempMemento
    }

    return Memento{}
}
```

However, our `Memento` list is replaced with a `Pop` method. It also returns a `memento` object but it will return them acting as a stack (last to enter, first to go out). So, we take the last element on the stack and store it in the `tempMemento` variable. Then we replace the stack with a new version that doesn't contain the last element on the next line. Finally, we return the `tempMemento` variable.

Until now, everything looks almost like in the previous example. We also talked about automating some tasks by using the Facade pattern, so let's do it. This is going to be called the `MementoFacade` type and will have the `SaveSettings` and `RestoreSettings` methods. The `SaveSettings` method takes a `Command`, stores it in an inner originator, and saves it in an inner `careTaker` field. The `RestoreSettings` method makes the opposite flow-restores an index of the `careTaker` and returns the `Command` inside the `Memento` object:

```
type MementoFacade struct {
    originator originator
    careTaker   careTaker
}

func (m *MementoFacade) SaveSettings(s Command) {
    m.originator.Command = s
    m.careTaker.Add(m.originator.NewMemento())
}

func (m *MementoFacade) RestoreSettings(i int) Command {
    m.originator.ExtractAndStoreCommand(m.careTaker.Memento(i))
    return m.originator.Command
}
```

Our Facade pattern will hold the contents of the originator and the care taker and will provide those two easy-to-use methods to save and restore settings.

So, how do we use this?

```
func main() {
    m := MementoFacade{}

    m.SaveSettings(Volume(4))
    m.SaveSettings(Mute(false))
```

First, we get a variable with a Facade pattern. Zero-value initialization will give us zero-valued `originator` and `caretaker` objects. They don't have any unexpected field so everything will initialize correctly (if any of them had a pointer, for example, it would be initialized to `nil` as mentioned in the *Zero initialization* section of Chapter 1, *Ready... Steady... Go!.*).

We create a `Volume` value with `Volume(4)` and, yes, we have used parentheses. The `Volume` type does not have any inner field like structs so we cannot use curly braces to set its value. The way to set it is to use parentheses (or create a pointer to the type `Volume` and then set the value of the pointed space). We also save a value of the type `Mute` using the Facade pattern.

We don't know what `Command` type is returned here, so we need to make a type assertion. We will make a small function to help us with this that checks the type and prints an appropriate value:

```
func assertAndPrint(c Command) {
    switch cast := c.(type) {
    case Volume:
        fmt.Printf("Volume:\t%d\n", cast)
    case Mute:
        fmt.Printf("Mute:\t%t\n", cast)
    }
}
```

The `assertAndPrint` method takes a `Command` type and casts it to the two possible types—`Volume` or `Mute`. In each case, it prints a message to the console with a personalized message. Now we can continue and finish the `main` function, which will look like this:

```
func main() {
    m := MementoFacade{}

    m.SaveSettings(Volume(4))
    m.SaveSettings(Mute(false))
```

```
    assertAndPrint(m.RestoreSettings(0))
    assertAndPrint(m.RestoreSettings(1))
}
```

The part highlighted in bold shows the new changes within the `main` function. We took the index 0 from the `careTaker` object and passed it to the new function and the same with the index 1. Running this small program, we should get the `Volume` and `Mute` values on the console:

```
$ go run memento_command.go
Mute:  false
Volume: 4
```

Great! In this small example, we have combined three different design patterns to keep getting comfortable using various patterns. Keep in mind that we could have abstracted the creation of `Volume` and `Mute` states to a Factory pattern too so this is not where would stop.

Last words on the Memento pattern

With the Memento pattern, we have learned a powerful way to create undoable operations that are very useful when writing UI applications but also when you have to develop transactional operations. In any case, the situation is the same: you need a `Memento`, an `Originator`, and a `caretaker` actor.



A **transaction operation** is a set of atomic operations that must all be done or fail. In other words, if you have a transaction composed of five operations and just one of them fails, the transaction cannot be completed and every modification done by the other four must be undone.

Interpreter design pattern

Now we are going to dig into a quite complex pattern. The **Interpreter** pattern is, in fact, widely used to solve business cases where it's useful to have a language to perform common operations. Let's see what we mean by language.

Description

The most famous interpreter we can talk about is probably SQL. It's defined as a special-purpose programming language for managing data held in relational databases. SQL is quite complex and big but, all in all, is a set of words and operators that allow us to perform operations such as insert, select, or delete.

Another typical example is musical notation. It's a language itself and the interpreter is the musician who knows the connection between a note and its representation on the instrument they are playing.

In computer science, it can be useful to design a small language for a variety of reasons: repetitive tasks, higher-level languages for non-developers, or **Interface Definition Languages (IDL)** such as **Protocol buffers** or **Apache Thrift**.

Objectives

Designing a new language, big or small, can be a time consuming task so it's very important to have the objectives clear before investing time and resources on writing an interpreter of it:

- Provide syntax for very common operations in some scope (such as playing notes).
- Have an intermediate language to translate actions between two systems. For example, the apps that generate the **Gcode** needed to print with 3D printers.
- Ease the use of some operations in an easier-to-use syntax.

SQL allows the use of relational databases in a very easy-to-use syntax (that can become incredibly complex too) but the idea is to not need to write your own functions to make insertions and searches.

Example - a polish notation calculator

A very typical example of an interpreter is to create a reverse polish notation calculator. For those who don't know what polish notation is, it's a mathematical notation to make operations where you write your operation first (sum) and then the values (3 4), so $+ 3 4$ is equivalent to the more common $3 + 4$ and its result would be 7. So, for a reverse polish notation, you put first the values and then the operation, so $3 4 +$ would also be 7.

Acceptance criteria for the calculator

For our calculator, the acceptance criteria we should pass to consider it done are as follows:

1. Create a language that allows making common arithmetic operations (sums, subtractions, multiplications, and divisions). The syntax is `sum` for sums, `mul` for multiplications, `sub` for subtractions, and `div` for divisions.
2. It must be done using reverse polish notation.
3. The user must be able to write as many operations in a row as they want.
4. The operations must be performed from left to right.

So the `3 4 sum 2 sub` notation is the same than $(3 + 4) - 2$ and result would be 5.

Unit test of some operations

In this case, we will only have a public method called `Calculate` that takes an operation with its values defined as a string and will return a value or an error:

```
func Calculate(o string) (int, error) {
    return 0, fmt.Errorf("Not implemented yet")
}
```

So, we will send a string like `"3 4 +"` to the `Calculate` method and it should return `7, nil`. Two tests more will check the correct implementation:

```
func TestCalculate(t *testing.T) {
    tempOperation = "3 4 sum 2 sub"
    res, err = Calculate(tempOperation)
    if err != nil {
        t.Error(err)
    }

    if res != 5 {
        t.Errorf("Expected result not found: %d != %d\n", 5, res)
    }
}
```

First, we are going to make the operation we have used as an example. The `3 4 sum 2 sub` notation is part of our language and we use it in the `Calculate` function. If an error is returned, the test fails. Finally, the result must be equal to 5 and we check it on the last lines. The next test checks the rest of the operators on slightly more complex operations:

```
tempOperation := "5 3 sub 8 mul 4 sum 5 div"
res, err := Calculate(tempOperation)
```

```
if err != nil {
    t.Error(err)
}

if res != 4 {
    t.Errorf("Expected result not found: %d != %d\n", 4, res)
}
}
```

Here, we repeated the preceding process with a longer operation, the $((5 - 3) * 8) + 4) / 5$ notation which is equal to 4. From left to right, it would be as follows:

```
((5 - 3) * 8) + 4) / 5
((2 * 8) + 4) / 5
(16 + 4) / 5
20 / 5
4
```

The test must fail, of course!

```
$ go test -v .
interpreter_test.go:9: Not implemented yet
interpreter_test.go:13: Expected result not found: 4 != 0
interpreter_test.go:19: Not implemented yet
interpreter_test.go:23: Expected result not found: 5 != 0
exit status 1
FAIL
```

Implementation

Implementation is going to be longer than testing this time. To start, we will define our possible operators in constants:

```
const (
    SUM = "sum"
    SUB = "sub"
    MUL = "mul"
    DIV = "div"
)
```

Interpreter patterns are usually implemented using an abstract syntax tree, something that is commonly achieved using a stack. We have created stacks before during the book so this should be already familiar to readers:

```
type polishNotationStack []int
```

```
func (p *polishNotationStack) Push(s int) {
    *p = append(*p, s)
}

func (p *polishNotationStack) Pop() int {
    length := len(*p)

    if length > 0 {
        temp := (*p)[length-1]
        *p = (*p)[:length-1]
        return temp
    }

    return 0
}
```

We have two methods--the `Push` method to add elements to the top of the stack and the `Pop` method to remove elements and return them. In case you are thinking that the line `*p = (*p) [:length-1]` is a bit cryptic, we'll explain it.

The value stored in the direction of `p` will be overridden with the actual value in the direction of `p` `(*p)` but taking only the elements from the beginning to the penultimate element of the array `(:length-1)`.

So, now we will go step by step with the `Calculate` function, creating more functions as far as we need them:

```
func Calculate(o string) (int, error) {
    stack := polishNotationStack{}
    operators := strings.Split(o, " ")
```

The first two things we need to do are to create the stack and to get all different symbols from the incoming operation (in this case, we aren't checking that it isn't empty). We split the incoming string operations by the space to get a nice slice of symbols (values and operators).

Next, we will iterate over every symbol by using range but we need a function to know whether the incoming symbol is a value or an operator:

```
func isOperator(o string) bool {
    if o == SUM || o == SUB || o == MUL || o == DIV {
        return true
    }

    return false
}
```

If the incoming symbol is any of the ones defined in our constants, the incoming symbol is an operator:

```
func Calculate(o string) (int, error) {
    stack := polishNotationStack{}
    operators := strings.Split(o, " ")

    for _, operatorString := range operators {
        if isOperator(operatorString) {
            right := stack.Pop()
            left := stack.Pop()

        } else {
            //Is a value
        }
    }
}
```

If it is an operator, we consider that we have already passed two values so what we have to do is to take those two values from the stack. The first value taken would be the rightmost and the second the leftmost (remember that in subtractions and divisions, the order of the operands is important). Then, we need some function to get the operation we want to perform:

```
func getOperationFunc(o string) func(a, b int) int {
    switch o {
    case SUM:
        return func(a, b int) int {
            return a + b
        }
    case SUB:
        return func(a, b int) int {
            return a - b
        }
    case MUL:
        return func(a, b int) int {
            return a * b
        }
    case DIV:
        return func(a, b int) int {
            return a / b
        }
    }
    return nil
}
```

The `getOperationFunc` function returns a two-argument function that returns an integer. We check the incoming operator and we return an anonymous function that performs the specified operation. So, now our `for` range continues like this:

```
func Calculate(o string) (int, error) {
    stack := polishNotationStack{}
    operators := strings.Split(o, " ")

    for _, operatorString := range operators {
        if isOperator(operatorString) {
            right := stack.Pop()
            left := stack.Pop()
            mathFunc := getOperationFunc(operatorString)      res :=
            mathFunc(left, right)      stack.Push(res)
        } else {
            //Is a value
        }
    }
}
```

The `mathFunc` variable is returned by the function. We use it immediately to perform the operation on the left and right values taken from the stack and we store its result in a new variable called `res`. Finally, we need to push this new value to the stack to keep operating with it later.

Now, here is the implementation when the incoming symbol is a value:

```
func Calculate(o string) (int, error) {
    stack := polishNotationStack{}
    operators := strings.Split(o, " ")

    for _, operatorString := range operators {
        if isOperator(operatorString) {
            right := stack.Pop()
            left := stack.Pop()
            mathFunc := getOperationFunc(operatorString)
            res := mathFunc(left, right)
            stack.Push(res)
        } else {
            val, err := strconv.Atoi(operatorString)
            if err != nil {
                return 0, err
            }
            stack.Push(val)
        }
    }
}
```

What we need to do every time we get a symbol is to push it to the stack. We have to parse the string symbol to a usable `int` type. This is commonly done with the `strconv` package by using its `Atoi` function. The `Atoi` function takes a string and returns an integer from it or an error. If everything goes well, the value is pushed into the stack.

At the end of the `range` statement, just one value must be stored on it, so we just need to return it and the function is done:

```
func Calculate(o string) (int, error) {
    stack := polishNotationStack{}
    operators := strings.Split(o, " ")

    for _, operatorString := range operators {
        if isOperator(operatorString) {
            right := stack.Pop()
            left := stack.Pop()
            mathFunc := getOperationFunc(operatorString)
            res := mathFunc(left, right)
            stack.Push(res)
        } else {
            val, err := strconv.Atoi(operatorString)
            if err != nil {
                return 0, err
            }

            stack.Push(val)
        }
    }
    return int(stack.Pop()), nil
}
```

Time to run the tests again:

```
$ go test -v .
ok
```

Great! We have just created a reverse polish notation interpreter in a very simple and easy way (we still lack the parser, but that's another story).

Complexity with the Interpreter design pattern

In this example, we haven't used any interfaces. This is not exactly how the Interpreter design pattern is defined in more object-oriented languages. However, this example is the simplest example possible to understand the objectives of the language and the next level is inevitably much more complex and not intended for beginner users.

With a more complex example, we will have to define a type containing more types of itself, a value, or nothing. With a parser, you create this abstract syntax tree to interpret it later.

The same example, done by using interfaces, would be as in the following description section.

Interpreter pattern again - now using interfaces

The main interface we are going to use is called the `Interpreter` interface. This interface has a `Read()` method that every symbol (value or operator) must implement:

```
type Interpreter interface {
    Read() int
}
```

We will implement only the sum and the subtraction from the operators and a type called `Value` for the numbers:

```
type Value int

func (v *Value) Read() int {
    return int(*v)
}
```

The `Value` is a type `int` that, when implementing the `Read` method, just returns its value:

```
type operationSum struct {
    Left  Interpreter
    Right Interpreter
}

func (a *operationSum) Read() int {
    return a.Left.Read() + a.Right.Read()
}
```

The `operationSum` struct has the `Left` and `Right` fields and its `Read` method returns the sum of each of their `Read` methods. The `operationSubtract` struct is the same but subtracting:

```
type operationSubtract struct {
    Left  Interpreter
    Right Interpreter
}

func (s *operationSubtract) Read() int {
```

```
        return s.Left.Read() - s.Right.Read()
    }
```

We also need a Factory pattern to create operators; we will call it the `operatorFactory` method. The difference now is that it not only accepts the symbol but also the `Left` and `Right` values taken from the stack:

```
func operatorFactory(o string, left, right Interpreter) Interpreter {
    switch o {
    case SUM:
        return &operationSum{
            Left: left,
            Right: right,
        }
    case SUB:
        return &operationSubtract{
            Left: left,
            Right: right,
        }
    }

    return nil
}
```

As we have just mentioned, we also need a stack. We can reuse the one from the previous example by changing its type:

```
type polishNotationStack []Interpreter

func (p *polishNotationStack) Push(s Interpreter) {
    *p = append(*p, s)
}

func (p *polishNotationStack) Pop() Interpreter {
    length := len(*p)

    if length > 0 {
        temp := (*p)[length-1]
        *p = (*p)[:length-1]
        return temp
    }

    return nil
}
```

Now the stack works with Interpreter pointers instead of `int` but its functionality is the same. Finally, our `main` method also looks similar to our previous example:

```
func main() {
    stack := polishNotationStack{}
    operators := strings.Split("3 4 sum 2 sub", " ")

    for _, operatorString := range operators {
        if operatorString == SUM || operatorString == SUB {
            right := stack.Pop()
            left := stack.Pop()
            mathFunc := operatorFactory(operatorString, left, right)
            res := value(mathFunc.Read())
            stack.Push(&res)
        } else {
            val, err := strconv.Atoi(operatorString)
            if err != nil {
                panic(err)
            }

            temp := value(val)
            stack.Push(&temp)
        }
    }

    println(int(stack.Pop().Read()))
}
```

Like before, we check whether the symbol is operator or value first. When it's a value, it pushes it into the stack.

When the symbol is an operator, we also take the right and left values from the stack, we call the Factory pattern using the current operator and the left and right values that we just took from the stack. Once we have the operator type, we just need to call its `Read` method to push the returned value to the stack too.

Finally, just one example must be left on the stack, so we print it:

```
$ go run interpreter.go
```

The power of the Interpreter pattern

This pattern is extremely powerful but it must also be used carefully. To create a language, it generates a strong coupling between its users and the functionality it provides. One can fall into the error of trying to create a too flexible language that is incredibly complex to use and maintain. Also, one can create a fairly small and useful language that doesn't interpret correctly sometimes and it could be a pain for its users.

In our example, we have omitted quite a lot of error-checking to focus on the implementation of the Interpreter. However, you'll need quite a lot of error checking and verbose output on errors to help the user correct its syntax errors. So, have fun writing your language but be nice to your users.

Summary

This chapter has dealt with three extremely powerful patterns that require a lot of practice before using them in production code. It's a very good idea to make some exercises with them by simulating typical production problems:

- Create a simple REST server that reuses most of the error-checking and connection functionality to provide an easy-to-use interface to practice the Template pattern
- Make a small library that can write to different databases but only in the case that all writes were OK, or delete the newly created writes to practice Memento for example
- Write your own language, to make simple things such as answering simple questions like bots usually do so you can practice a bit of the Interpreter pattern

The idea is to practice coding and reread any section until you get comfortable with each pattern.

7

Behavioral Patterns - Visitor, State, Mediator, and Observer Design Patterns

This is the last chapter about Behavioral patterns and it also closes this book's section about common, well known design patterns in Go language.

In this chapter, we are going to look at three more design patterns. Visitor pattern is very useful when you want to abstract away some functionality from a set of objects.

State is used commonly to build **Finite State Machines (FSM)** and, in this section, we will develop a small *guess the number* game.

Finally, the Observer pattern is commonly used in event-driven architectures and is gaining a lot of traction again, especially in the microservices world.

After this chapter, we'll need to feel very comfortable with common design patterns before digging in concurrency and the advantages (and complexity), it brings to design patterns.

Visitor design pattern

In the next design pattern, we are going to delegate some logic of an object's type to an external type called the visitor that will visit our object to perform operations on it.

Description

In the Visitor design pattern, we are trying to separate the logic needed to work with a specific object outside of the object itself. So we could have many different visitors that do some things to specific types.

For example, imagine that we have a log writer that writes to console. We could make the logger "visitble" so that you can prepend any text to each log. We could write a Visitor pattern that prepends the date, the time, and the hostname to a field stored in the object.

Objectives

With Behavioral design patterns we are mainly dealing with algorithms. Visitor patterns are not an exception. The objectives that we are trying to achieve are as follows:

- To separate the algorithm of some type from its implementation within some other type
- To improve the flexibility of some types by using them with little or no logic at all so all new functionality can be added without altering the object structure
- To fix a structure or behavior that would break the open/closed principle in a type

You might be thinking what the open/closed principle is. In computer science, the open/closed principle states that: *entities should be open for extension but closed for modification*. This simple state has lots of implications that allows building more maintainable software and less prone to errors. And the Visitor pattern helps us to delegate some commonly changing algorithm from a type that we need it to be "stable" to an external type that can change often without affecting our original one.

A log appender

We are going to develop a simple log appender as an example of the Visitor pattern. Following the approach we have had in previous chapters, we will start with an extremely simple example to clearly understand how the Visitor design pattern works before jumping to a more complex one. We have already developed similar examples modifying texts too, but in slightly different ways.

For this particular example, we will create a Visitor that appends different information to the types it "visits".

Acceptance criteria

To effectively use the Visitor design pattern, we must have two roles--a visitor and a visitable. The `Visitor` is the type that will act within a `Visitable` type. So a `Visitable` interface implementation has an algorithm detached to the `Visitor` type:

1. We need two message loggers: `MessageA` and `MessageB` that will print a message with an `A:` or a `B:` respectively before the message.
2. We need a `Visitor` able to modify the message to be printed. It will append the text "Visited A" or "Visited B" to them, respectively.

Unit tests

As we mentioned before, we will need a role for the `Visitor` and the `Visitable` interfaces. They will be interfaces. We also need the `MessageA` and `MessageB` structs:

```
package visitor

import (
    "io"
    "os"
    "fmt"
)

type MessageA struct {
    Msg string
    Output io.Writer
}

type MessageB struct {
    Msg string
    Output io.Writer
}

type Visitor interface {
    VisitA(*MessageA)
    VisitB(*MessageB)
}

type Visitable interface {
    Accept(Visitor)
}

type MessageVisitor struct { }
```

The types `MessageA` and `MessageB` structs both have an `Msg` field to store the text that they will print. The output `io.Writer` will implement the `os.Stdout` interface by default or a new `io.Writer` interface, like the one we will use to check that the contents are correct.

The `Visitor` interface has a `Visit` method, one for each of `Visitable` interface's `MessageA` and `MessageB` type. The `Visitable` interface has a method called `Accept(Visitor)` that will execute the decoupled algorithm.

Like in previous examples, we will create a type that implements the `io.Writer` package so that we can use it in tests:

```
package visitor

import "testing"

type TestHelper struct {
    Received string
}

func (t *TestHelper) Write(p []byte) (int, error) {
    t.Received = string(p)
    return len(p), nil
}
```

The `TestHelper` struct implements the `io.Writer` interface. Its functionality is quite simple; it stores the written bytes on the `Received` field. Later we can check the contents of `Received` to test against our expected value.

We will write just one test that will check the overall correctness of the code. Within this test, we will write two sub tests: one for `MessageA` and one for `MessageB` types:

```
func Test_Overall(t *testing.T) {
    testHelper := &TestHelper{}
    visitor := &MessageVisitor{}
    ...
}
```

We will use a `TestHelper` struct and a `MessageVisitor` struct on each test for each message type. First, we will test the `MessageA` type:

```
func Test_Overall(t *testing.T) {
    testHelper := &TestHelper{}
    visitor := &MessageVisitor{}

    t.Run("MessageA test", func(t *testing.T) {
        msg := MessageA{
```

```
    Msg: "Hello World",
    Output: testHelper,
}

msg.Accept(visitor)
msg.Print()

expected := "A: Hello World (Visited A)"
if testHelper.Received != expected {
    t.Errorf("Expected result was incorrect. %s != %s",
        testHelper.Received, expected)
}
})
...
}
```

This is the full first test. We created `MessageA` struct, giving it a value `Hello World` for the `Msg` field and the pointer to `TestHelper`, which we created at the beginning of the test. Then, we execute its `Accept` method. Inside the `Accept(Visitor)` method on the `MessageA` struct, the `VisitA(*MessageA)` method is executed to alter the contents of the `Msg` field (that's why we passed the pointer to `VisitA` method, without a pointer the contents won't be persisted).

To test if the `visitor` type has done its job within the `Accept` method, we must call the `Print()` method on the `MessageA` type later. This way, the `MessageA` struct must write the contents of `Msg` to the provided `io.Writer` interface (our `TestHelper`).

The last part of the test is the check. According to the description of *acceptance criteria 2*, the output text of `MessageA` type must be prefixed with the text `A:`, the stored message and the text `" (Visited) "` just at the end. So, for the `MessageA` type, the expected text must be `"A: Hello World (Visited) "`, this is the check that we did in the `if` section.

The `MessageB` type has a very similar implementation:

```
t.Run("MessageB test", func(t *testing.T) {
    msg := MessageB {
        Msg: "Hello World",
        Output: testHelper,
    }

    msg.Accept(visitor)
    msg.Print()

    expected := "B: Hello World (Visited B)"
    if testHelper.Received != expected {
        t.Errorf("Expected result was incorrect. %s != %s",
            testHelper.Received, expected)
    }
})
```

```
        testHelper.Received, expected)
    }
}
}
```

In fact, we have just changed the type from `MessageA` to `MessageB` and the expected text now is "B: Hello World (Visited B)". The `Msg` field is also "Hello World" and we also used the `TestHelper` type.

We still lack the correct implementations of the interfaces to compile the code and run the tests. The `MessageA` and `MessageB` structs have to implement the `Accept(Visitor)` method:

```
func (m *MessageA) Accept(v Visitor) {
    //Do nothing
}

func (m *MessageB) Accept(v Visitor) {
    //Do nothing
}
```

We need the implementations of the `VisitA(*MessageA)` and `VisitB(*MessageB)` methods that are declared on the `Visitor` interface. The `MessageVisitor` interface is the type that must implement them:

```
func (mf *MessageVisitor) VisitA(m *MessageA) {
    //Do nothing
}
func (mf *MessageVisitor) VisitB(m *MessageB) {
    //Do nothing
}
```

Finally, we will create a `Print()` method for each message type. This is the method that we will use to test the contents of the `Msg` field on each type:

```
func (m *MessageA) Print() {
    //Do nothing
}

func (m *MessageB) Print() {
    //Do nothing
}
```

Now we can run the tests to really check if they are failing yet:

```
go test -v .
==== RUN    Test_Overall
```

```
==== RUN Test_Overall/MessageA_test
==== RUN Test_Overall/MessageB_test
--- FAIL: Test_Overall (0.00s)
    --- FAIL: Test_Overall/MessageA_test (0.00s)
        visitor_test.go:30: Expected result was incorrect. != A: Hello
World (Visited A)
    --- FAIL: Test_Overall/MessageB_test (0.00s)
        visitor_test.go:46: Expected result was incorrect. != B: Hello
World (Visited B)
FAIL
exit status 1
FAIL
```

The outputs of the tests are clear. The expected messages were incorrect because the contents were empty. It's time to create the implementations.

Implementation of Visitor pattern

We will start completing the implementation of the `VisitA(*MessageA)` and `VisitB(*MessageB)` methods:

```
func (mf *MessageVisitor) VisitA(m *MessageA) {
    m.Msg = fmt.Sprintf("%s %s", m.Msg, "(Visited A)")
}
func (mf *MessageVisitor) VisitB(m *MessageB) {
    m.Msg = fmt.Sprintf("%s %s", m.Msg, "(Visited B)")
}
```

Its functionality is quite straightforward--the `fmt.Sprintf` method returns a formatted string with the actual contents of `m.Msg`, a white space, and the message, `Visited`. This string will be stored on the `Msg` field, overriding the previous contents.

Now we will develop the `Accept` method for each message type that must execute the corresponding Visitor:

```
func (m *MessageA) Accept(v Visitor) {
    v.VisitA(m)
}

func (m *MessageB) Accept(v Visitor) {
    v.VisitB(m)
}
```

This small code has some implications on it. In both cases, we are using a `Visitor`, which in our example is exactly the same as the `MessageVisitor` interface, but they could be completely different. The key is to understand that the Visitor pattern executes an algorithm in its `Visit` method that deals with the `Visitable` object. What could the `Visitor` be doing? In this example, it alters the `Visitable` object, but it could be simply fetching information from it. For example, we could have a `Person` type with lots of fields: name, surname, age, address, city, postal code, and so on. We could write a `Visitor` to fetch just the name and surname from a person as a unique string, a visitor to fetch the address info for a different section of an app, and so on.

Finally, there is the `Print()` method, which will help us to test the types. We mentioned before that it must print to the `Stdout` call by default:

```
func (m *MessageA) Print() {
    if m.Output == nil {
        m.Output = os.Stdout
    }

    fmt.Fprintf(m.Output, "A: %s", m.Msg)
}

func (m *MessageB) Print() {
    if m.Output == nil {
        m.Output = os.Stdout
    }
    fmt.Fprintf(m.Output, "B: %s", m.Msg)
}
```

It first checks the content of the `Output` field to assign the output of the `os.Stdout` call in case it is null. In our tests, we are storing a pointer there to our `TestHelper` type so this line is never executed in our test. Finally, each message type prints to the `Output` field, the full message stored in the `Msg` field. This is done by using the `Fprintf` method, which takes an `io.Writer` package as the first argument and the text to format as the next arguments.

Our implementation is now complete and we can run the tests again to see if they all pass now:

```
go test -v .
==== RUN  Test_Overall
==== RUN  Test_Overall/MessageA_test
==== RUN  Test_Overall/MessageB_test
--- PASS: Test_Overall (0.00s)
    --- PASS: Test_Overall/MessageA_test (0.00s)
    --- PASS: Test_Overall/MessageB_test (0.00s)
PASS
```

ok

Everything is OK! The Visitor pattern has done its job flawlessly and the message contents were altered after calling their `Visit` methods. The very important thing here is that we can add more functionality to both the structs, `MessageA` and `MessageB`, without altering their types. We can just create a new Visitor type that does everything on the `Visitable`, for example, we can create a `Visitor` to add a method that prints the contents of the `Msg` field:

```
type MsgFieldVisitorPrinter struct {}

func (mf *MsgFieldVisitorPrinter) VisitA(m *MessageA) {
    fmt.Printf(m.Msg)
}

func (mf *MsgFieldVisitorPrinter) VisitB(m *MessageB) {
    fmt.Printf(m.Msg)
}
```

We have just added some functionality to both types without altering their contents! That's the power of the Visitor design pattern.

Another example

We will develop a second example, this one a bit more complex. In this case, we will emulate an online shop with a few products. The products will have plain types, with just fields and we will make a couple of visitors to deal with them in the group.

First of all, we will develop the interfaces. The `ProductInfoRetriever` type has a method to get the price and the name of the product. The `Visitor` interface, like before, has a `Visit` method that accepts the `ProductInfoRetriever` type. Finally, `Visitable` interface is exactly the same; it has an `Accept` method that takes a `Visitor` type as an argument:

```
type ProductInfoRetriever interface {
    GetPrice() float32
    GetName() string
}

type Visitor interface {
    Visit(ProductInfoRetriever)
}

type Visitable interface {
    Accept(Visitor)
}
```

All the products of the online shop must implement the `ProductInfoRetriever` type. Also, most products will have some common fields, such as name or price (the ones defined in the `ProductInfoRetriever` interface). We created the `Product` type, implemented the `ProductInfoRetriever` and the `Visitable` interfaces, and embedded it on each product:

```
type Product struct {
    Price float32
    Name  string
}

func (p *Product) GetPrice() float32 {
    return p.Price
}

func (p *Product) Accept(v Visitor) {
    v.Visit(p)
}

func (p *Product) GetName() string {
    return p.Name
}
```

Now we have a very generic `Product` type that can store the information about almost any product of the shop. For example, we could have a `Rice` and a `Pasta` product:

```
type Rice struct {
    Product
}

type Pasta struct {
    Product
}
```

Each has the `Product` type embedded. Now we need to create a couple of `Visitors` interfaces, one that sums the price of all products and one that prints the name of each product:

```
type PriceVisitor struct {
    Sum float32
}

func (pv *PriceVisitor) Visit(p ProductInfoRetriever) {
    pv.Sum += p.GetPrice()
}

type NamePrinter struct {
```

```
    ProductList string
}

func (n *NamePrinter) Visit(p ProductInfoRetriever) {
    n.Names = fmt.Sprintf("%s\n%s", p.GetName(), n.ProductList)
}
```

The PriceVisitor struct takes the value of the Price variable of the ProductInfoRetriever type, passed as an argument, and adds it to the Sum field. The NamePrinter struct stores the name of the ProductInfoRetriever type, passed as an argument, and appends it to a new line on the ProductList field.

Time for the main function:

```
func main() {
    products := make([]Visitable, 2)
    products[0] = &Rice{
        Product: Product{
            Price: 32.0,
            Name: "Some rice",
        },
    }
    products[1] = &Pasta{
        Product: Product{
            Price: 40.0,
            Name: "Some pasta",
        },
    }

    //Print the sum of prices
    priceVisitor := &PriceVisitor{}

    for _, p := range products {
        p.Accept(priceVisitor)
    }

    fmt.Printf("Total: %f\n", priceVisitor.Sum)

    //Print the products list
    nameVisitor := &NamePrinter{}

    for _, p := range products {
        p.Accept(nameVisitor)
    }

    fmt.Printf("\nProduct list:\n-----\n%s",
        nameVisitor.ProductList)
```

}

We create a slice of two `Visitable` objects: a `Rice` and a `Pasta` type with some arbitrary names. Then we iterate for each of them using a `PriceVisitor` instance as an argument. We print the total price after the range for. Finally, we repeat this operation with the `NamePrinter` and print the resulting `ProductList`. The output of this `main` function is as follows:

```
go run visitor.go
Total: 72.000000
Product list:
-----
Some pasta
Some rice
```

Ok, this is a nice example of the Visitor pattern but... what if there are special considerations about a product? For example, what if we need to sum 20 to the total price of a `Fridge` type? OK, let's write the `Fridge` structure:

```
type Fridge struct {
    Product
}
```

The idea here is to just override the `GetPrice()` method to return the product's price plus 20:

```
type Fridge struct {
    Product
}

func (f *Fridge) GetPrice() float32 {
    return f.Product.Price + 20
}
```

Unfortunately, this isn't enough for our example. The `Fridge` structure is not of a `Visitable` type. The `Product` struct is of a `Visitable` type and the `Fridge` struct has a `Product` struct embedded but, as we mentioned in earlier chapters, a type that embeds a second type cannot be considered of that latter type, even when it has all its fields and methods. The solution is to also implement the `Accept(Visitor)` method so that it can be considered as a `Visitable`:

```
type Fridge struct {
    Product
}

func (f *Fridge) GetPrice() float32 {
```

```
    return f.Product.Price + 20
}

func (f *Fridge) Accept(v Visitor) {
    v.Visit(f)
}
```

Let's rewrite the `main` function to add this new `Fridge` product to the slice:

```
func main() {
    products := make([]Visitable, 3)
    products[0] = &Rice{
        Product: Product{
            Price: 32.0,
            Name: "Some rice",
        },
    }
    products[1] = &Pasta{
        Product: Product{
            Price: 40.0,
            Name: "Some pasta",
        },
    }
    products[2] = &Fridge{
        Product: Product{
            Price: 50,
            Name: "A fridge",
        },
    }
    ...
}
```

Everything else continues the same. Running this new `main` function produces the following output:

```
$ go run visitor.go
Total: 142.000000
Product list:
-----
A fridge
Some pasta
Some rice
```

As expected, the total price is higher now, outputting the sum of the rice (32), the pasta (40), and the fridge (50 of the product plus 20 of the transport, so 70). We could be adding visitors forever to this products, but the idea is clear--we decoupled some algorithms outside of the types to the visitors.

Visitors to the rescue!

We have seen a powerful abstraction to add new algorithms to some types. However, because of the lack of overloading in Go, this pattern could be limiting in some aspects (we have seen it in the first example, where we had to create the `VisitA` and `VisitB` implementations). In the second example, we haven't dealt with this limitation because we have used an interface to the `Visit` method of the `Visitor` struct, but we just used one type of visitor (`ProductInfoRetriever`) and we would have the same problem if we implemented a `Visit` method for a second type, which is one of the objectives of the original *Gang of Four* design patterns.

State design pattern

State patterns are directly related to FSMs. An FSM, in very simple terms, is something that has one or more states and travels between them to execute some behaviors. Let's see how the State pattern helps us to define FSM.

Description

A light switch is a common example of an FSM. It has two states--on and off. One state can transit to the other and vice versa. The way that the State pattern works is similar. We have a `State` interface and an implementation of each state we want to achieve. There is also usually a context that holds cross-information between the states.

With FSM, we can achieve very complex behaviors by splitting their scope between states. This way we can model pipelines of execution based on any kind of inputs or create event-driven software that responds to particular events in specified ways.

Objectives

The main objectives of the State pattern is to develop FSM are as follows:

- To have a type that alters its own behavior when some internal things have changed
- Model complex graphs and pipelines can be upgraded easily by adding more states and rerouting their output states

A small guess the number game

We are going to develop a very simple game that uses FSM. This game is a number guessing game. The idea is simple--we will have to guess some number between 0 and 10 and we have just a few attempts or we'll lose.

We will leave the player to choose the level of difficulty by asking how many tries the user has before losing. Then, we will ask the player for the correct number and keep asking if they don't guess it or if the number of tries reaches zero.

Acceptance criteria

For this simple game, we have five acceptance criteria that basically describe the mechanics of the game:

1. The game will ask the player how many tries they will have before losing the game.
2. The number to guess must be between 0 and 10.
3. Every time a player enters a number to guess, the number of retries drops by one.
4. If the number of retries reaches zero and the number is still incorrect, the game finishes and the player has lost.
5. If the player guesses the number, the player wins.

Implementation of State pattern

The idea of unit tests is quite straightforward in a State pattern so we will spend more time explaining in detail the mechanism to use it, which is a bit more complex than usual.

First of all, we need the interface to represent the different states and a game context to store the information between states. For this game, the context needs to store the number of retries, if the user has won or not, the secret number to guess, and the current state. The state will have an `executeState` method that accepts one of these contexts and returns `true` if the game has finished, or `false` if not:

```
type GameState interface {
    executeState(*GameContext) bool
}

type GameContext struct {
    SecretNumber int
    Retries     int
}
```

```
    Won bool
    Next GameState
}
```

As described in *acceptance criteria 1*, the player must be able to introduce the number of retries they want. This will be achieved by a state called `StartState`. Also, the `StartState` struct must prepare the game, setting the context to its initial value before the player:

```
type StartState struct{}
func(s *StartState) executeState(c *GameContext) bool {
    c.Next = &AskState{}

    rand.Seed(time.Now().UnixNano())
    c.SecretNumber = rand.Intn(10)

    fmt.Println("Introduce a number a number of retries to set the
difficulty:")
    fmt.Fscanf(os.Stdin, "%d\n", &c.Retries)

    return true
}
```

First of all, the `StartState` struct implements the `GameState` structure because it has the `executeState(*Context)` method of Boolean type on its structure. At the beginning of this state, it sets the only state possible after executing this one--the `AskState` state. The `AskState` struct is not declared yet, but it will be the state where we ask the player for a number to guess.

In the next two lines, we use the `Rand` package of Go to generate a random number. In the first line, we feed the random generator with the `int64` type number returned by the current moment, so we ensure a random feed in each execution (if you put a constant number here, the randomizer will also generate the same number too). The `rand.Intn(int)` method returns an integer number between zero and the specified number, so here we cover *acceptance criteria 2*.

Next, a message asking for a number of retries to set precedes the `fmt.Fscanf` method, a powerful function where you can pass it an `io.Reader` (the standard input of the console), a format (number), and an interface to store the contents of the reader, in this case, the `Retries` field of the context.

Finally, we return `true` to tell the engine that the game must continue. Let's see the `AskState` struct, which we have used at the beginning of the function:

```
type AskState struct {}
func (a *AskState) executeState(c *GameContext) bool{
    fmt.Printf("Introduce a number between 0 and 10, you have %d tries
left\n", c.Retries)

    var n int
    fmt.Fscanf(os.Stdin, "%d", &n)
    c.Retries = c.Retries - 1

    if n == c.SecretNumber {
        c.Won = true
        c.Next = &FinishState{}
    }

    if c.Retries == 0 {
        c.Next = &FinishState{}
    }

    return true
}
```

The `AskState` structure also implements the `GameState` state, as you have probably guessed already. This states starts with a message for the player, asking them to insert a new number. In the next three lines, we create a local variable to store the contents of the number that the player will introduce. We used the `fmt.Fscanf` method again, as we did in `StartState` struct to capture the player's input and store it in the variable `n`. Then, we have one retry less in our counter, so we have to subtract one to the number of retries represented in the `Retries` field.

Then, there are two checks: one that checks if the user has entered the correct number, in which case the context field `Won` is set to `true` and the next state is set to the `FinishState` struct (not declared yet).

The second check is controlling that the number of retries has not reached zero, in which case it won't let the player ask again for a number and it will send the player to the `FinishState` struct directly. After all, we have to tell the game engine again that the game must continue by returning `true` in the `executeState` method.

Finally, we define the `FinishState` struct. It controls the exit status of the game, checking the contents of the `Won` field in the context object:

```
type FinishState struct{}
```

```
func(f *FinishState) executeState(c *GameContext) bool {
    if c.Won {
        println("Congrats, you won")
    } else {
        println("You lose")
    }
    return false
}
```

The `TheFinishState` struct also implements the `GameState` state by having `executeState` method in its structure. The idea here is very simple--if the player has won (this field is set previously in the `AskState` struct), the `FinishState` structure will print the message `Congrats, you won`. If the player has not won (remember that the zero value of the Boolean variable is `false`), the `FinishState` prints the message `You lose`.

In this case, the game can be considered finished, so we return `false` to say that the game must not continue.

We just need the `main` method to play our game:

```
func main() {
    start := StartState{}
    game := GameContext{
        Next:&start,
    }
    for game.Next.executeState(&game) {}
}
```

Well, yes, it can't be simpler. The game must begin with the `start` method, although it could be abstracted more outside in case that the game needs more initialization in the future, but in our case it is fine. Then, we create a context where we set the `Next` state as a pointer to the `start` variable. So the first state that will be executed in the game will be the `StartState` state.

The last line of the `main` function has a lot of things just there. We create a loop, without any statement inside it. As with any loop, it keeps looping after the condition is not satisfied. The condition we are using is the returned value of the `GameState`s structure, `true` as soon as the game is not finished.

So, the idea is simple: we execute the state in the context, passing a pointer to the context to it. Each state returns `true` until the game has finished and the `FinishState` struct will return `false`. So our for loop will keep looping, waiting for a `false` condition sent by the `FinishState` structure to end the application.

Let's play once:

```
go run state.go
Introduce a number a number of retries to set the difficulty:
5
Introduce a number between 0 and 10, you have 5 tries left
8
Introduce a number between 0 and 10, you have 4 tries left
2
Introduce a number between 0 and 10, you have 3 tries left
1
Introduce a number between 0 and 10, you have 2 tries left
3
Introduce a number between 0 and 10, you have 1 tries left
4
You lose
```

We lost! We set the number of retries to 5. Then we kept inserting numbers, trying to guess the secret number. We entered 8, 2, 1, 3, and 4, but it wasn't any of them. I don't even know what the correct number was; let's fix this!

Go to the definition of the `FinishState` struct and change the line where it says `You lose`, and replace it with the following:

```
fmt.Printf("You lose. The correct number was: %d\n", c.SecretNumber)
```

Now it will show the correct number. Let's play again:

```
go run state.go
Introduce a number a number of retries to set the difficulty:
3
Introduce a number between 0 and 10, you have 3 tries left
6
Introduce a number between 0 and 10, you have 2 tries left
2
Introduce a number between 0 and 10, you have 1 tries left
1
You lose. The correct number was: 9
```

This time we make it a little harder by setting only three tries... and we lost again. I entered 6, 2, and 1, but the correct number was 9. Last try:

```
go run state.go
Introduce a number a number of retries to set the difficulty:
5
Introduce a number between 0 and 10, you have 5 tries left
3
Introduce a number between 0 and 10, you have 4 tries left
```

```
4
Introduce a number between 0 and 10, you have 3 tries left
5
Introduce a number between 0 and 10, you have 2 tries left
6
Congrats, you won
```

Great! This time we lowered the difficulty, allowing up to five tries and we won! we even had one more try left, but we guessed the number in the fourth try after entering 3, 4, 5. The correct number was 6, which was my fourth try.

A state to win and a state to lose

Have you realized that we could have a winning and a lose state instead of printing the messages directly in the `FinishState` struct? This way we could, for example, check some hypothetical scoreboard in the win section to see if we have set a record. Let's refactor our game. First we need a `WinState` and a `LoseState` struct:

```
type WinState struct{ }

func (w *WinState) executeState(c *GameContext) bool {
    println("Congrats, you won")

    return false
}

type LoseState struct{ }

func (l *LoseState) executeState(c *GameContext) bool {
    fmt.Printf("You lose. The correct number was: %d\n", c.SecretNumber)
    return false
}
```

These two new states have nothing new. They contain the same messages that were previously in the `FinishState` state that, by the way, must be modified to use these new states:

```
func (f *FinishState) executeState(c *GameContext) bool {
    if c.Won {
        c.Next = &WinState{}
    } else {
        c.Next = &LoseState{}
    }
    return true
}
```

Now, the finish state doesn't print anything and, instead, delegates this to the next state in the chain--the `WinState` structure, if the user has won and the `LoseState` struct, if not. Remember that the game doesn't finish on the `FinishState` struct now, and we must return `true` instead of `false` to notify to the engine that it must keep executing states in the chain.

The game built using the State pattern

You must be thinking now that you can extend this game forever with new states, and it's true. The power of the State pattern is not only the capacity to create a complex FSM, but also the flexibility to improve it as much as you want by adding new states and modifying some old states to point to the new ones without affecting the rest of the FSM.

Mediator design pattern

Let's continue with the Mediator pattern. As its name implies, it's a pattern that will be in between two types to exchange information. But, why will we want this behavior at all? Let's look at this in detail.

Description

One of the key objectives of any design pattern is to avoid tight coupling between objects. This can be done in many ways, as we have seen already.

But one particularly effective method when the application grows a lot is the Mediator pattern. The Mediator pattern is the perfect example of a pattern that is commonly used by every programmer without thinking very much about it.

Mediator pattern will act as the type in charge of exchanging communication between two objects. This way, the communicating objects don't need to know each other and can change more freely. The pattern that maintains which objects give what information is the Mediator.

Objectives

As previously described, the main objectives of the Mediator pattern are about loose coupling and encapsulation. The objectives are:

- To provide loose coupling between two objects that must communicate between them
- To reduce the amount of dependencies of a particular type to the minimum by passing these needs to the Mediator pattern

A calculator

For the Mediator pattern, we are going to develop an extremely simple arithmetic calculator. You're probably thinking that a calculator is so simple that it does not need any pattern. But we will see that this is not exactly true.

Our calculator will only do two very simple operations: sum and subtract.

Acceptance criteria

It sounds quite funny to talk about acceptance criteria to define a calculator, but let's do it anyway:

1. Define an operation called `Sum` that takes a number and adds it to another number.
2. Define an operation called `Subtract` that takes a number and subtracts it to another number.

Well, I don't know about you, but I really need a rest after this *complex* criteria. So why are we defining this so much? Patience, you will have the answer soon.

Implementation

We have to jump directly to the implementation because we cannot test that the sum will be correct (well, we can, but we will be testing if Go is correctly written!). We could test that we pass the acceptance criteria, but it's a bit of an overkill for our example.

So let's start by implementing the necessary types:

```
package main

type One struct{}
type Two struct{}
type Three struct{}
type Four struct{}
```

```
type Five struct{}
type Six struct{}
type Seven struct{}
type Eight struct{}
type Nine struct{}
type Zero struct{}
```

Well... this look quite awkward. We already have numeric types in Go to perform these operations, we don't need a type for each number!

But let's continue for a second with this insane approach. Let's implement the `One` struct:

```
type One struct{}

func (o *One) OnePlus(n interface{}) interface{} {
    switch n.(type) {
    case One:
        return &Two{}
    case Two:
        return &Three{}
    case Three:
        return &Four{}
    case Four:
        return &Five{}
    case Five:
        return &Six{}
    case Six:
        return &Seven{}
    case Seven:
        return &Eight{}
    case Eight:
        return &Nine{}
    case Nine:
        return [2]interface{}{&One{}, &Zero{}}
    default:
        return fmt.Errorf("Number not found")
    }
}
```

OK , I'll stop here. What is wrong with this implementation? This is completely crazy! It's overkill to make every operation possible between numbers to make sums! Especially when we have more than one digit.

Well, believe it or not, this is how a lot of software is commonly designed today. A small app where an object uses two or three objects grows, and it ends up using dozens of them. It becomes an absolute hell to simply add or remove a type from the application because it is hidden in some of this craziness.

So what can we do in this calculator? Use a Mediator type that frees all the cases:

```
func Sum(a, b interface{}) interface{}{
    switch a := a.(type) {
        case One:
            switch b.(type) {
                case One:
                    return &Two{}
                case Two:
                    return &Three{}
                default:
                    return fmt.Errorf("Number not found")
            }
        case Two:
            switch b.(type) {
                case One:
                    return &Three{}
                case Two:
                    return &Four{}
                default:
                    return fmt.Errorf("Number not found")
            }
        case int:
            switch b := b.(type) {
                case One:
                    return &Three{}
                case Two:
                    return &Four{}
                case int:
                    return a + b
                default:
                    return fmt.Errorf("Number not found")
            }
        default:
            return fmt.Errorf("Number not found")
    }
}
```

We have just developed a couple of numbers to keep things short. The `Sum` function acts as a mediator between two numbers. First it checks the type of the first number named `a`. Then, for each type of the first number, it checks the type of the second number named `b` and returns the resulting type.

While the solution still looks very crazy now, the only one that knows about all possible numbers in the calculator is the `Sum` function. But take a closer look and you'll see that we have added a type case for the `int` type. We have cases `One`, `Two`, and `int`. Inside the `int` case, we also have another `int` case for the `b` number. What do we do here? If both types are of the `int` case, we can return the sum of them.

Do you think that this will work? Let's write a simple `main` function:

```
func main() {
    fmt.Printf("%#v\n", Sum(One{}, Two{}))
    fmt.Printf("%d\n", Sum(1, 2))
}
```

We print the sum of type `One` and type `Two`. By using the `"%#v"` format, we ask to print information about the type. The second line in the function uses `int` types, and we also print the result. This in the console produces the following output:

```
$go run mediator.go
&main.Three{}
7
```

Not very impressive, right? But let's think for a second. By using the Mediator pattern, we have been able to refactor the initial calculator, where we have to define every operation on every type to a Mediator pattern, the `Sum` function.

The nice thing is that, thanks to the Mediator pattern, we have been able to start using integers as values for our calculator. We have just defined the simplest example by adding two integers, but we could have done the same with an integer and the `type`:

```
case One:
    switch b := b.(type) {
    case One:
        return &Two{}
    case Two:
        return &Three{}
    case int:
        return b+1
    default:
        return fmt.Errorf("Number not found")
    }
```

With this small modification, we can now use type `One` with an `int` as number `b`. If we keep working on this Mediator pattern, we could achieve a lot of flexibility between types, without having to implement every possible operation between them, generating a tight coupling.

We'll add a new `Sum` method in the main function to see this in action:

```
func main() {
    fmt.Printf("%#v\n", Sum(One{}, Two{}))
    fmt.Printf("%d\n", Sum(1,2))
    fmt.Printf("%d\n", Sum(One{},2))
}
$go run mediator.go&main.Three{}33
```

Nice. The Mediator pattern is in charge of knowing about the possible types and returns the most convenient type for our case, which is an integer. Now we could keep growing this `Sum` function until we completely get rid of using the numeric types we have defined.

Uncoupling two types with the Mediator

We have carried out a disruptive example to try to think outside the box and reason deeply about the Mediator pattern. Tight coupling between entities in an app can become really complex to deal with in the future and allow more difficult refactoring if needed.

Just remember that the Mediator pattern is there to act as a managing type between two types that don't know about each other so that you can take one of the types without affecting the other and replace a type in a more easy and convenient way.

Observer design pattern

We will finish the common *Gang of Four* design patterns with my favorite: the Observer pattern, also known as publish/subscriber or publish/listener. With the State pattern, we defined our first event-driven architecture, but with the Observer pattern we will really reach a new level of abstraction.

Description

The idea behind the Observer pattern is simple--to subscribe to some event that will trigger some behavior on many subscribed types. Why is this so interesting? Because we uncouple an event from its possible handlers.

For example, imagine a login button. We could code that when the user clicks the button, the button color changes, an action is executed, and a form check is performed in the background. But with the Observer pattern, the type that changes the color will subscribe to the event of the clicking of the button. The type that checks the form and the type that performs an action will subscribe to this event too.

Objectives

The Observer pattern is especially useful to achieve many actions that are triggered on one event. It is also especially useful when you don't know how many actions are performed after an event in advance or there is a possibility that the number of actions is going to grow in the near future. To resume, do the following:

- Provide an event-driven architecture where one event can trigger one or more actions
- Uncouple the actions that are performed from the event that triggers them
- Provide more than one event that triggers the same action

The notifier

We will develop the simplest possible application to fully understand the roots of the Observer pattern. We are going to make a `Publisher` struct, which is the one that triggers an event so it must accept new observers and remove them if necessary. When the `Publisher` struct is triggered, it must notify all its observers of the new event with the data associated.

Acceptance criteria

The requirements must tell us to have some type that triggers some method in one or more actions:

1. We must have a publisher with a `NotifyObservers` method that accepts a message as an argument and triggers a `Notify` method on every observer subscribed.
2. We must have a method to add new subscribers to the publisher.
3. We must have a method to remove new subscribers from the publisher.

Unit tests

Maybe you have realized that our requirements defined almost exclusively the Publisher type. This is because the action performed by the observer is irrelevant for the Observer pattern. It should simply execute an action, in this case the `Notify` method, that one or many types will implement. So let's define this only interface for this pattern:

```
type Observer interface {
    Notify(string)
}
```

The `Observer` interface has a `Notify` method that accepts a `string` type that will contain the message to spread. It does not need to return anything, but we could return an error if we want to check if all observers have been reached when calling the `publish` method of the Publisher structure.

To test all the acceptance criteria, we just need a structure called `Publisher` with three methods:

```
type Publisher struct {
    ObserversList []Observer
}

func (s *Publisher) AddObserver(o Observer) {}

func (s *Publisher) RemoveObserver(o Observer) {}

func (s *Publisher) NotifyObservers(m string) {}
```

The `Publisher` structure stores the list of subscribed observers in a slice field called `ObserversList`. Then it has the three methods mentioned on the acceptance criteria-the `AddObserver` method to subscribe a new observer to the publisher, the `RemoveObserver` method to unsubscribe an observer, and the `NotifyObservers` method with a `string` that acts as the message we want to spread between all observers.

With these three methods, we have to set up a root test to configure the `Publisher` and three subtests to test each method. We also need to define a test type structure that implements the `Observer` interface. This structure is going to be called `TestObserver`:

```
type TestObserver struct {
    ID      int
    Message string
}

func (p *TestObserver) Notify(m string) {
    fmt.Printf("Observer %d: message '%s' received \n", p.ID, m)
```

```
    p.Message = m  
}
```

The `TestObserver` structure implements the Observer pattern by defining a `Notify(string)` method in its structure. In this case, it prints the received message together with its own observer ID. Then, it stores the message in its `Message` field. This allows us to check later if the content of the `Message` field is as expected. Remember that it could also be done by passing the `testing.T` pointer and the expected message and checking within the `TestObserver` structure.

Now we can set up the `Publisher` structure to execute the three tests. We will create three instances of the `TestObserver` structure:

```
func TestSubject(t *testing.T) {  
    testObserver1 := &TestObserver{1, ""}  
    testObserver2 := &TestObserver{2, ""}  
    testObserver3 := &TestObserver{3, ""}  
    publisher := Publisher{}
```

We have given a different ID to each observer so that we can see later that each of them has printed the expected message. Then, we have added the observers by calling the `AddObserver` method on the `Publisher` structure.

Let's write an `AddObserver` test, it must add a new observer to the `ObserversList` field of the `Publisher` structure:

```
t.Run("AddObserver", func(t *testing.T) {  
    publisher.AddObserver(testObserver1)  
    publisher.AddObserver(testObserver2)  
    publisher.AddObserver(testObserver3)  
  
    if len(publisher.ObserversList) != 3 {  
        t.Fail()  
    }  
})
```

We have added three observers to the `Publisher` structure, so the length of the slice must be 3. If it's not 3, the test will fail.

The `RemoveObserver` test will take the observer with ID 2 and remove it from the list:

```
t.Run("RemoveObserver", func(t *testing.T) {  
    publisher.RemoveObserver(testObserver2)  
  
    if len(publisher.ObserversList) != 2 {  
        t.Errorf("The size of the observer list is not the " +
```

```
    "expected. 3 != %d\n", len(publisher.ObserversList))
}
for _, observer := range publisher.ObserversList {
    testObserver, ok := observer.(TestObserver)
    if !ok {
        t.Fail()
    }
    if testObserver.ID == 2 {
        t.Fail()
    }
}
})
```

After removing the second observer, the length of the Publisher structure must be 2 now. We also check that none of the observers left have the ID 2 because it must be removed.

The last method to test is the `Notify` method. When using the `Notify` method, all instances of `TestObserver` structure must change their `Message` field from empty to the passed message (`Hello World!` in this case). First we will check that all the `Message` fields are, in fact, empty before calling the `NotifyObservers` test:

```
t.Run("Notify", func(t *testing.T) {
    for _, observer := range publisher.ObserversList {
        printObserver, ok := observer.(*TestObserver)
        if !ok {
            t.Fail()
            break
        }

        if printObserver.Message != "" {
            t.Errorf("The observer's Message field weren't " + " empty: %s\n",
printObserver.Message)
        }
    }
})
```

Using a `for` statement, we are iterating over the `ObservesList` field to slice in the `publisher` instance. We need to make a type casting from a pointer to an observer, to a pointer to the `TestObserver` structure, and check that the casting has been done correctly. Then, we check that the `Message` field is actually empty.

The next step is to create a message to send--in this case, it will be `"Hello World!"` and then pass this message to the `NotifyObservers` method to notify every observer on the list (currently observers 1 and 3 only):

```
...
message := "Hello World!"
```

```
    publisher.NotifyObservers(message)

    for _, observer := range publisher.ObserversList {
        printObserver, ok := observer.(*TestObserver)
        if !ok {
            t.Fail()
            break
        }

        if printObserver.Message != message {
            t.Errorf("Expected message on observer %d was " +
                "not expected: '%s' != '%s'\n", printObserver.ID,
                printObserver.Message, message)
        }
    }
}
```

After calling the `NotifyObservers` method, each `TestObserver` tests in the `ObserversList` field must have the message "Hello World!" stored in their `Message` field. Again, we use a `for` loop to iterate over every observer of the `ObserversList` field and we typecast each to a `TestObserver` test (remember that `TestObserver` structure doesn't have any field as it's an interface). We could avoid type casting by adding a new `Message()` method to `Observer` instance and implementing it in the `TestObserver` structure to return the contents of the `Message` field. Both methods are equally valid. Once we have type casted to a `TestObserver` method called `printObserver` variable as a local variable, we check that each instance in the `ObserversList` structure has the string "Hello World!" stored in their `Message` field.

Time to run the tests that must fail all to check their effectiveness in the later implementation:

```
go test -v
==== RUN  TestSubject
==== RUN  TestSubject/AddObserver
==== RUN  TestSubject/RemoveObserver
==== RUN  TestSubject/Notify
---- FAIL: TestSubject (0.00s)
    --- FAIL: TestSubject/AddObserver (0.00s)
    --- FAIL: TestSubject/RemoveObserver (0.00s)
        observer_test.go:40: The size of the observer list is not the
expected. 3 != 0
    --- PASS: TestSubject/Notify (0.00s)
FAIL
exit status 1
FAIL
```

Something isn't working as expected. How is the `Notify` method passing the tests if we haven't implemented the function yet? Take a look at the test of the `Notify` method again. The test iterates over the `ObserversList` structure and each `Fail` call is inside this for loop. If the list is empty, it won't iterate, so it won't execute any `Fail` call.

Let's fix this issue by adding a small non-empty list check at the beginning of the `Notify` test:

```
if len(publisher.ObserversList) == 0 {  
    t.Errorf("The list is empty. Nothing to test\n")  
}
```

And we will rerun the tests to see if the `TestSubject/Notify` method is already failing:

```
go test -v  
==== RUN TestSubject  
==== RUN TestSubject/AddObserver  
==== RUN TestSubject/RemoveObserver  
==== RUN TestSubject/Notify  
---- FAIL: TestSubject (0.00s)  
---- FAIL: TestSubject/AddObserver (0.00s)  
---- FAIL: TestSubject/RemoveObserver (0.00s)  
        observer_test.go:40: The size of the observer list is not the  
expected. 3 != 0  
---- FAIL: TestSubject/Notify (0.00s)  
        observer_test.go:58: The list is empty. Nothing to test  
FAIL  
exit status 1  
FAIL
```

Nice, all of them are failing and now we have some guarantee on our tests. We can proceed to the implementation.

Implementation

Our implementation is just to define the `AddObserver`, the `RemoveObserver`, and the `NotifyObservers` methods:

```
func (s *Publisher) AddObserver(o Observer) {  
    s.ObserversList = append(s.ObserversList, o)  
}
```

The AddObserver method adds the Observer instance to the ObserversList structure by appending the pointer to the current list of pointers. This one was very easy. The AddObserver test must be passing now (but not the rest or we could have done something wrong):

```
go test -v
==== RUN TestSubject
==== RUN TestSubject/AddObserver
==== RUN TestSubject/RemoveObserver
==== RUN TestSubject/Notify
---- FAIL: TestSubject (0.00s)
    --- PASS: TestSubject/AddObserver (0.00s)
    --- FAIL: TestSubject/RemoveObserver (0.00s)
        observer_test.go:40: The size of the observer list is not the
expected. 3 != 3
        --- FAIL: TestSubject/Notify (0.00s)
            observer_test.go:87: Expected message on observer 1 was not
expected: 'default' != 'Hello World!'
            observer_test.go:87: Expected message on observer 2 was not
expected: 'default' != 'Hello World!'
            observer_test.go:87: Expected message on observer 3 was not
expected: 'default' != 'Hello World!'
FAIL
exit status 1
FAIL
```

Excellent. Just the AddObserver method has passed the test, so we can now continue to the RemoveObserver method:

```
func (s *Publisher) RemoveObserver(o Observer) {
    var indexToRemove int

    for i, observer := range s.ObserversList {
        if observer == o {
            indexToRemove = i
            break
        }
    }

    s.ObserversList = append(s.ObserversList[:indexToRemove],
    s.ObserversList[indexToRemove+1:]...)
}
```

The RemoveObserver method will iterate for each element in the `ObserversList` structure, comparing the `Observer` object's `o` variable with the ones stored in the list. If it finds a match, it saves the index in the local variable, `indexToRemove`, and stops the iteration. The way to remove indexes on a slice in Go is a bit tricky:

1. First, we need to use slice indexing to return a new slice containing every object from the beginning of the slice to the index we want to remove (not included).
2. Then, we get another slice from the index we want to remove (not included) to the last object in the slice
3. Finally, we join the previous two new slices into a new one (the `append` function)

For example, in a list from 1 to 10 in which we want to remove the number 5, we have to create a new slice, joining a slice from 1 to 4 and a slice from 6 to 10.

This index removal is done with the `append` function again because we are actually appending two lists together. Just take a closer look at the three dots at the end of the second argument of the `append` function. The `append` function adds an element (the second argument) to a slice (the first), but we want to append an entire list. This can be achieved using the three dots, which translate to something like *keep adding elements until you finish the second array*.

Ok, let's run this test now:

```
go test -v
==== RUN  TestSubject
==== RUN  TestSubject/AddObserver
==== RUN  TestSubject/RemoveObserver
==== RUN  TestSubject/Notify
---- FAIL: TestSubject (0.00s)
---- PASS: TestSubject/AddObserver (0.00s)
---- PASS: TestSubject/RemoveObserver (0.00s)
---- FAIL: TestSubject/Notify (0.00s)
        observer_test.go:87: Expected message on observer 1 was not
        expected: 'default' != 'Hello World!'
        observer_test.go:87: Expected message on observer 3 was not
        expected: 'default' != 'Hello World!'
FAIL
exit status 1
FAIL
```

We continue in the good path. The `RemoveObserver` test has been fixed without fixing anything else. Now we have to finish our implementation by defining the `NotifyObservers` method:

```
func (s *Publisher) NotifyObservers(m string) {
```

```
fmt.Printf("Publisher received message '%s' to notify observers\n", m)
for _, observer := range s.ObserversList {
    observer.Notify(m)
}
}
```

The `NotifyObservers` method is quite simple because it prints a message to the console to announce that a particular message is going to be passed to the `Observers`. After this, we use a for loop to iterate over `ObserversList` structure and execute each `Notify(string)` method by passing the argument `m`. After executing this, all observers must have the message `Hello World!` stored in their `Message` field. Let's see if this is true by running the tests:

```
go test -v
==== RUN  TestSubject
==== RUN  TestSubject/AddObserver
==== RUN  TestSubject/RemoveObserver
==== RUN  TestSubject/Notify
Publisher received message 'Hello World!' to notify observers
Observer 1: message 'Hello World!' received
Observer 3: message 'Hello World!' received
--- PASS: TestSubject (0.00s)
    --- PASS: TestSubject/AddObserver (0.00s)
    --- PASS: TestSubject/RemoveObserver (0.00s)
    --- PASS: TestSubject/Notify (0.00s)
PASS
ok
```

Excellent! We can also see the outputs of the `Publisher` and `Observer` types on the console. The `Publisher` structure prints the following message:

```
hey! I have received the message 'Hello World!' and I'm going to pass the
same message to the observers
```

After this, all observers print their respective messages as follows:

```
hey, I'm observer 1 and I have received the message 'Hello World!'
```

And the same for the third observer.

Summary

We have unlocked the power of event-driven architectures with the State pattern and the Observer pattern. Now you can really execute asynchronous algorithms and operations in your application that respond to events in your system.

The Observer pattern is commonly used in UI's. Android programming is filled with Observer patterns so that the Android SDK can delegate the actions to be performed by the programmers creating an app.

8

Introduction to Go Concurrency

We have just finished with the *Gang Of Four* design patterns that are commonly used in object oriented programming languages. They have been used extensively for the last few decades (even before they were explicitly defined in a book).

In this chapter, we are going to see concurrency in the Go language. We will, learn that with multiple cores and multiple processes, applications can help us to achieve better performance and endless possibilities. We will look at how to use some of the already known patterns in concurrently safe ways.

A little bit of history and theory

When we talk about Go's concurrency, it's impossible not to talk about history. In the last decades, we saw an improvement in the speed of CPUs until we reached the hardware limits imposed by current hardware materials, design, and architectures. When we reached this point, we started to play with the first multicore computers, the first double CPU motherboards, and then single CPUs with more than one core in their heart.

Unfortunately, the languages we are using are still the ones created when we had single core CPUs, such as Java or C++. While being terrific systems languages, they lack a proper concurrency support by design. You can develop concurrent apps in both of the languages used in your project by using third party tools or by developing your own (not a very easy task).

Go's concurrency was designed with these caveats in mind. The creators wanted garbage collected and procedural language that is familiar for newcomers, but which, at the same time, can be used to write concurrent applications easily and without affecting the core of the language.

We have experienced this in the early chapters. We have developed more than 20 design patterns without a word about concurrency. This clearly shows that the concurrent features of the Go language are completely separated from the core language while being part of it, a perfect example of abstraction and encapsulation.

There are many concurrency models in computer science, the most famous being the actor model present in languages such as **Erlang** or **Scala**. Go, on the other side, uses **Communicating Sequential Processes (CSP)**, which has a different approach to concurrency.

Concurrency versus parallelism

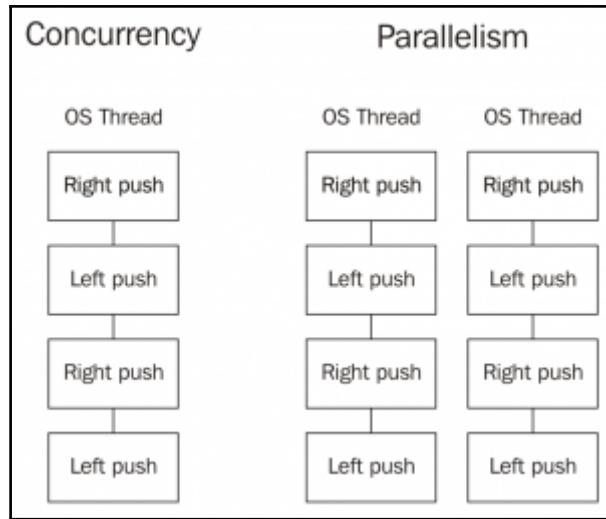
Many people have misunderstood the differences between both, even thinking that they are the same. There is a popular speech by Rob Pike, one of the creators of Go, *Concurrency is not parallelism*, which I really agree with. As a quick summary of the talk, we can extract the following:

- Concurrency is about dealing with many things at once
- Parallelism is about doing many things at the same time

Concurrency enables parallelism by designing a correct structure of concurrency work.

For example, we can think of the mechanism of a bike. When we pedal, we usually push down the pedal to produce force (and this push, raises our opposite leg on the opposite pedal). We cannot push with both legs at the same time because the cranks don't allow us to do it. But this design allows the construction of a parallel bike, commonly called a **tandem bike**. A tandem bike is a bike that two people can ride at the same time; they both pedal and apply force to the bike.

In the bike example, concurrency is the design of a bike that, with two legs (Goroutines), you can produce power to move the bike by yourself. The design is concurrent and correct. If we use a tandem bike and two people (two cores), the solution is concurrent, correct, and parallel. But the key thing is that with a concurrent design, we don't have to worry about parallelism; we can think about it as an extra feature if our concurrent design is correct. In fact, we can use the tandem bike with only one person, but the concurrent design of the legs, pedals, chain, wheels of a bike is still correct.



With concurrency, on the left side, we have a design and a structure that is executed sequentially by the same CPU core. Once we have this design and structure, parallelism can be achieved by simply repeating this structure on a different thread.

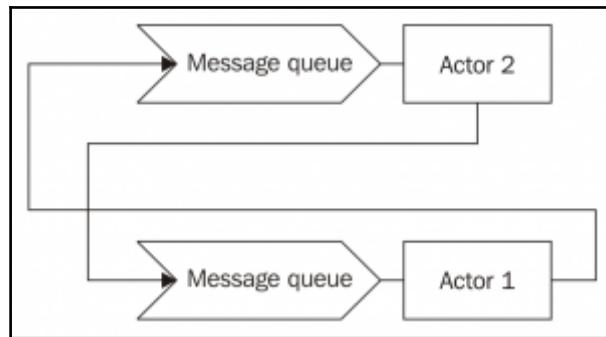
This is how Go eases the reasoning about concurrent and parallel programs by simply not worrying too much about parallel execution and focusing much more on concurrent design and structure. Breaking a big task into smaller tasks that can be run concurrently usually provides much better performance in a single-core computer, but, if this design can also be run in parallel, we could achieve an even higher throughput (or not, depending on the design).

In fact, we can set the number of cores in use in a Go app by setting the environment variable `GOMAXPROCS` to the number of cores we want. This is not only useful when using schedulers, such as **Apache Mesos**, but it gives us more control about how a Go app works and performs.

So, to recap, it is very important to keep in mind that concurrency is about structure and parallelism is about execution. We must think about making our programs concurrent in a better way, by breaking them down into smaller pieces of work, and Go's scheduler will try to make them parallel if it's possible and allowed.

CSP versus actor-based concurrency

The most common and, perhaps, intuitive way to think about concurrency is close to the way the actor model works.



In the actor model, if **Actor 1** wants to communicate with **Actor 2**, then **Actor 1** must know **Actor 2** first; for example, it must have its process ID, maybe from the creation step, and put a message on its inbox queue. After placing the message, **Actor 1** can continue its tasks without getting blocked if **Actor 2** cannot process the message immediately.

CSP, on the other side, introduces a new entity into the equation-channels. Channels are the way to communicate between processes because they are completely anonymous (unlike actors, where we need to know their process IDs). In the case of CSP, we don't have a process ID to use to communicate. Instead, we have to create a channel to the processes to allow incoming and outgoing communication. In this case, what we know that the receiver is the channel it uses to receive data:



In this diagram, we can see that the processes are anonymous, but we have a channel with ID 1, that is, **Channel 1**, which connects them together. This abstraction does not tell us how many processes are on each side of the channel; it simply connects them and allows communication between processes by using the channel.

The key here is that channels isolate both extremes so that process A can send data through a channel that will be handled by potentially one or more processes that' are transparent to A. It also works the same in reverse; process B can receive data from many channels one at a time.

Goroutines

In Go, we achieve concurrency by working with Goroutines. They are like processes that run applications in a computer concurrently; in fact, the main loop of Go could be considered a Goroutine, too. Goroutines are used in places where we would use actors. They execute some logic and die (or keep looping if necessary).

But Goroutines are not threads. We can launch thousands of concurrent Goroutines, even millions. They are incredibly cheap, with a small growth stack. We will use Goroutines to execute code that we want to work concurrently. For example, three calls to three services to compose a response can be designed concurrently with three Goroutines to do the service calls potentially in parallel and a fourth Goroutine to receive them and compose the response. What's the point here? That if we have a computer with four cores, we could potentially run this service call in parallel, but if we use a one-core computer, the design will still be correct and the calls will be executed concurrently in only one core. By designing concurrent applications, we don't need to worry about parallel execution.

Returning to the bike analogy, we were pushing the pedals of the bike with our two legs. That's two Goroutines concurrently pushing the pedals. When we use the tandem, we had a total of four Goroutines, possibly working in parallel. But we also have two hands to handle the front and rear brakes. That's a total of eight Goroutines for our two threads bike. Actually, we don't pedal when we brake and we don't brake when we pedal; that's a correct concurrent design. Our nervous system transports the information about when to stop pedaling and when to start braking. In Go, our nervous system is composed of channels; we will see them after playing a bit with Goroutines first.

Our first Goroutine

Enough of the explanations now. Let's get our hands dirty. For our first Goroutine, we will print the message `Hello World!` in a Goroutine. Let's start with what we've been doing up until now:

```
package main

func main() {
    helloWorld()
}

func helloWorld() {
    println("Hello World!")
}
```

Running this small snippet of code will simply output Hello World! in the console:

```
$ go run main.go
Hello World!
```

Not impressive at all. To run it in a new Goroutine, we just need to add the keyword `go` at the beginning of the call to the function:

```
package main

func main() {
    go helloWorld()
}

func helloWorld(){
    println("Hello World!")
}
```

With this simple word, we are telling Go to start a new Goroutine running the contents of the `helloWorld` function.

So, let's run it:

```
$ go run main.go
$
```

What? It printed nothing! Why is that? Things get complicated when you start to deal with concurrent applications. The problem is that the `main` function finishes before the `helloWorld` function gets executed. Let's analyse it step by step. The `main` function starts and schedules a new Goroutine that will execute the `helloWorld` function, but the function isn't executed when the function finishes--it is still in the scheduling process.

So, our `main` problem is that the `main` function has to wait for the Goroutine to be executed before finishing. So let's pause for a second to give some room to the Goroutine:

```
package main
import "time"

func main() {
    go helloWorld()

    time.Sleep(time.Second)
}

func helloWorld(){
    println("Hello World!")
}
```

The `time.Sleep` function effectively sleeps the main Goroutine for one second before continuing (and exiting). If we run this now, we must get the message:

```
$ go run main.go
Hello World!
```

I suppose you must have noticed by now the small gap of time where the program is freezing before finishing. This is the function for sleeping. If you are doing a lot of tasks, you might want to raise the waiting time to whatever you want. Just remember that in any application the `main` function cannot finish before the rest of the Goroutines.

Anonymous functions launched as new Goroutines

We have defined the `helloWorld` function so that it can be launched with a different Goroutine. This is not strictly necessary because you can launch snippets of code directly in the function's scope:

```
package main
import "time"

func main() {
    go func() {
        println("Hello World")
    }()
    time.Sleep(time.Second)
}
```

This is also valid. We have used an anonymous function and we have launched it in a new Goroutine using the `go` keyword. Take a closer look at the closing braces of the function—they are followed by opening and closing parenthesis, indicating the execution of the function.

We can also pass data to anonymous functions:

```
package main
import "time"

func main() {
    go func(msg string) {
        println(msg)
    }("Hello World")
    time.Sleep(time.Second)
}
```

This is also valid. We had defined an anonymous function that received a string, which then printed the received string. When we called the function in a different Goroutine, we passed the message we wanted to print. In this sense, the following example would also be valid:

```
package main
import "time"

func main() {
    messagePrinter := func(msg string) {
        println(msg)
    }

    go messagePrinter("Hello World")
    go messagePrinter("Hello goroutine")
    time.Sleep(time.Second)
}
```

In this case, we have defined a function within the scope of our `main` function and stored it in a variable called `messagePrinter`. Now we can concurrently print as many messages as we want by using the `messagePrinter(string)` signature:

```
$ go run main.go
Hello World
Hello goroutine
```

We have just scratched the surface of concurrent programming in Go, but we can already see that it can be quite powerful. But we definitely have to do something with that sleeping period. WaitGroups can help us with this problem.

WaitGroups

`WaitGroup` comes in the synchronization package (the `sync` package) to help us synchronize many concurrent Goroutines. It works very easily--every time we have to wait for one Goroutine to finish, we add 1 to the group, and once all of them are added, we ask the group to wait. When the Goroutine finishes, it says `Done` and the `WaitGroup` will take one from the group:

```
package main

import (
    "sync"
    "fmt"
)

func main() {
```

```
var wait sync.WaitGroup
wait.Add(1)

go func() {
    fmt.Println("Hello World!")
    wait.Done()
}()
wait.Wait()
}
```

This is the simplest possible example of a WaitGroup. First, we created a variable to hold it called the `wait` variable. Next, before launching the new Goroutine, we say to the `WaitGroup` hey, you'll have to wait for one thing to finish by using the `wait.Add(1)` method. Now we can launch the 1 that the `WaitGroup` has to wait for, which in this case is the previous Goroutine that prints Hello World and says Done (by using the `wait.Done()` method) at the end of the Goroutine. Finally, we indicate to the `WaitGroup` to wait. We have to remember that the function `wait.Wait()` was probably executed before the Goroutine.

Let's run the code again:

```
$ go run main.go
Hello World!
```

Now it just waits the necessary time and not one millisecond more before exiting the application. Remember that when we use the `Add(value)` method, we add entities to the `WaitGroup`, and when we use the `Done()` method, we subtract one.

Actually, the `Add` function takes a delta value, so the following code is equivalent to the previous:

```
package main

import (
    "sync"
    "fmt"
)

func main() {
    var wait sync.WaitGroup
    wait.Add(1)

    go func() {
        fmt.Println("Hello World!")
        wait.Add(-1)
    }()
}
```

```
    wait.Wait()
}
```

In this case, we added 1 before launching the Goroutine and we added -1 (subtracted 1) at the end of it. If we know in advance how many Goroutines we are going to launch, we can also call the `Add` method just once:

```
package main
import (
    "fmt"
    "sync"
)

func main() {
    var wait sync.WaitGroup

    goRoutines := 5
    wait.Add(goRoutines)

    for i := 0; i < goRoutines; i++ {
        go func(goRoutineID int) {
            fmt.Printf("ID:%d: Hello goroutines!\n", goRoutineID)
            wait.Done()
        }(i)
    }
    wait.Wait()
}
```

In this example, we are going to create five Goroutines (as stated in the `goRoutines` variable). We know it in advance, so we simply add them all to the `WaitGroup`. We are then going to launch the same amount of `goroutine` variables by using a `for` loop. Every time one Goroutine finishes, it calls the `Done()` method of the `WaitGroup` that is effectively waiting at the end of the main loop.

Again, in this case, the code reaches the end of the `main` function before all Goroutines are launched (if any), and the `WaitGroup` makes the execution of the main flow wait until all `Done` messages are called. Let's run this small program:

```
$ go run main.go
ID:4: Hello goroutines!
ID:0: Hello goroutines!
ID:1: Hello goroutines!
ID:2: Hello goroutines!
ID:3: Hello goroutines!
```

We haven't mentioned it before, but we have passed the iteration index to each Goroutine as the parameter `GoroutineID` to print it with the message `Hello goroutines!` You might also have noticed that the Goroutines aren't executed in order. Of course! We are dealing with a scheduler that doesn't guarantee the order of execution of the Goroutines. This is something to keep in mind when programming concurrent applications. In fact, if we execute it again, we won't necessarily get the same order of output:

```
$ go run main.go
ID:4: Hello goroutines!
ID:2: Hello goroutines!
ID:1: Hello goroutines!
ID:3: Hello goroutines!
ID:0: Hello goroutines!
```

Callbacks

Now that we know how to use `WaitGroups`, we can also introduce the concept of callbacks. If you have ever worked with languages like JavaScript that use them extensively, this section will be familiar to you. A callback is an anonymous function that will be executed within the context of a different function.

For example, we want to write a function to convert a string to uppercase, as well as making it asynchronous. How do we write this function so that we can work with callbacks? There's a little trick—we can have a function that takes a string and returns a string:

```
func toUpperSync(word string) string {
    //Code will go here
}
```

So take the returning type of this function (a string) and put it as the second parameter in an anonymous function, as shown here:

```
func toUpperSync(word string, f func(string)) {
    //Code will go here
}
```

Now, the `toUpperSync` function returns nothing, but also takes a function that, by coincidence, also takes a string. We can execute this function with the result we will usually return.

```
func toUpperSync(word string, f func(string)) {
    f(strings.ToUpper(word))
}
```

We execute the `f` function with the result of calling the `strings.ToUpper` method with the provided word (which returns the word parameter in uppercase). Let's write the `main` function too:

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    toUpperSync("Hello Callbacks!", func(v string) {
        fmt.Printf("Callback: %s\n", v)
    })
}

func toUpperSync(word string, f func(string)) {
    f(strings.ToUpper(word))
}
```

In our main code, we have defined our callback. As you can see, we passed the test `Hello Callbacks!` to convert it to uppercase. Next we pass the callback to be executed with the result of passing our string to uppercase. In this case, we simply print the text in the console with the text `Callback` in front of it. When we execute this code, we get the following result:

```
$ go run main.go
Callback: HELLO CALLBACKS!
```

Strictly speaking, this is a synchronous callback. To make it asynchronous we have to introduce some concurrent handling:

```
package main
import (
    "fmt"
    "strings"
    "sync"
)

var wait sync.WaitGroup

func main() {
    wait.Add(1)

    toUpperAsync("Hello Callbacks!", func(v string) {
        fmt.Printf("Callback: %s\n", v)
        wait.Done()
    })
}
```

```
    }

    println("Waiting async response...")
    wait.Wait()
}

func toUpperAsync(word string, f func(string)) {
    go func() {
        f(strings.ToUpper(word))
    }()
}
```

This is the same code executed asynchronously. We use WaitGroups to handle concurrency (we will see later that channels can also be used for this). Now, our function `toUpperAsync` is, as its name implies, asynchronous. We launched the callback in a different Goroutine by using the keyword `go` when calling the callback. We write a small message to show the ordering nature of the concurrent execution more precisely. We wait until the callback signals that it's finished and we can exit the program safely. When we execute this, we get the following result:

```
$ go run main.go

Waiting async response...
Callback: HELLO CALLBACKS!
```

As you can see, the program reaches the end of the `main` function before executing the callback in the `toUpperAsync` function. This pattern brings many possibilities, but leaves us open to one big problem called **callback hell**.

Callback hell

The term **callback hell** is commonly used to refer to when many callbacks have been stacked within each other. This makes them difficult to reason with and handle when they grow too much. For example, using the same code as before, we could stack another asynchronous call with the contents that we previously printed to the console:

```
func main() {
    wait.Add(1)

    toUpperAsync("Hello Callbacks!", func(v string) {
        toUpperAsync(fmt.Sprintf("Callback: %s\n", v), func(v string) {
            fmt.Printf("Callback within %s", v)
            wait.Done()
        })
    })
}
```

```
    println("Waiting async response...")
    wait.Wait()
}
```

(We have omitted imports, the package name, and the `toUpperAsync` function as they have not changed.) Now we have the `toUpperAsync` function within a `toUpperAsync` function, and we could embed many more if we want. In this case, we again pass the text that we previously printed on the console to use it in the following callback. The inner callback finally prints it on the console, giving the following output:

```
$ go run main.go
Waiting async response...
Callback within CALLBACK: HELLO CALLBACKS!
```

In this case, we can assume that the outer callback will be executed before the inner one. That's why we don't need to add one more to the `WaitGroup`.

The point here is that we must be careful when using callbacks. In very complex systems, too many callbacks are hard to reason with and hard to deal with. But with care and rationality, they are powerful tools.

Mutexes

If you are working with concurrent applications, you have to deal with more than one resource potentially accessing some memory location. This is usually called **race condition**.

In simpler terms, a race condition is similar to that moment where two people try to get the last piece of pizza at exactly the same time--their hands collide. Replace the pizza with a variable and their hands with Goroutines and we'll have a perfect analogy.

There is one character at the dinner table to solve this issues--a father or mother. They have kept the pizza on a different table and we have to ask for permission to stand up before getting our slice of pizza. It doesn't matter if all the kids ask at the same time--they will only allow one kid to stand.

Well, a mutex is like our parents. They'll control who can access the pizza--I mean, a variable--and they won't allow anyone else to access it.

To use a mutex, we have to actively lock it; if it's already locked (another Goroutine is using it), we'll have to wait until it's unlocked again. Once we get access to the mutex, we can lock it again, do whatever modifications are needed, and unlock it again. We'll look at this using an example.

An example with mutexes - concurrent counter

Mutexes are widely used in concurrent programming. Maybe not so much in Go because it has a more idiomatic way of concurrent programming in its use of channels, but it's worth seeing how they work for the situations where channels simply don't fit so well.

For our example, we are going to develop a small concurrent counter. This counter will add one to an integer field in a `Counter` type. This should be done in a concurrent-safe way.

Our `Counter` structure is defined like this:

```
type Counter struct {
    sync.Mutex
    value int
}
```

The `Counter` structure has a field of `int` type that stores the current value of the count. It also embeds the `Mutex` type from the `sync` package. Embedding this field will allow us to lock and unlock the entire structure without actively calling a specific field.

Our `main` function launches 10 Goroutines that try to add one to the `value` field of `Counter` structure. All of this is done concurrently:

```
package main

import (
    "sync"
    "time"
)

func main() {
    counter := Counter{}

    for i := 0; i < 10; i++ {
        go func(i int) {
            counter.Lock()
            counter.value++
            defer counter.Unlock()
        }(i)
    }
    time.Sleep(time.Second)

    counter.Lock()
    defer counter.Unlock()

    println(counter.value)
}
```

We have created a type called `Counter`. Using a `for` loop, we have launched a total of 10 Goroutines, as we saw in the *Anonymous functions launched as new Goroutines* section. But inside every Goroutine, we are locking the counter so that no more Goroutines can access it, adding one to the field value, and unlocking it again so others can access it.

Finally, we'll print the value held by the counter. It must be 10 because we have launched 10 Goroutines.

But how can we know that this program is thread safe? Well, Go comes with a very handy built-in feature called the "race detector".

Presenting the race detector

We already know what a race condition is. To recap, it is used when two processes try to access the same resource at the same time with one or more writing operations (both processes writing or one process writing while the other reads) involved at that precise moment.

Go has a very handy tool to help diagnose race conditions, that you can run in your tests or your main application directly. So let's reuse the example we just wrote for the *mutexes* section and run it with the race detector. This is as simple as adding the `-race` command-line flag to the command execution of our program:

```
$ go run -race main.go  
10
```

Well, not very impressive is it? But in fact it is telling us that it has not detected a potential race condition in the code of this program. Let's make the detector of `-race` flag warn us of a possible race condition by not locking `counter` before we modify it:

```
for i := 0; i < 10; i++ {  
    go func(i int) {  
        //counter.Lock()  
        counter.value++  
        //counter.Unlock()  
    }(i)  
}
```

Inside the `for` loop, comment the `Lock` and `Unlock` calls before and after adding 1 to the field value. This will introduce a race condition. Let's run the same program again with the race flag activated:

```
$ go run -race main.go  
=====
```

```
WARNING: DATA RACE
Read at 0x00c42007a068 by goroutine 6:
  main.main.func1()
    [some_path]/concurrency/locks/main.go:19 +0x44
Previous write at 0x00c42007a068 by goroutine 5:
  main.main.func1()
    [some_path]/concurrency/locks/main.go:19 +0x60
Goroutine 6 (running) created at:
  main.main()
    [some_path]/concurrency/locks/main.go:21 +0xb6
Goroutine 5 (finished) created at:
  main.main()
    [some_path]/concurrency/locks/main.go:21 +0xb6
=====
10
Found 1 data race(s)
exit status 66
```

I have reduced the output a bit to see things more clearly. We can see a big, uppercase message reading `WARNING: DATA RACE`. But this output is very easy to reason with. First, it is telling us that some memory position represented by *line 19* on our `main.go` file is reading some variable. But there is also a write operation in *line 19* of the same file!

This is because a "`++`" operation requires a read of the current value and a write to add one to it. That's why the race condition is in the same line, because every time it's executed it reads and writes the field in the `Counter` structure.

But let's keep in mind that the race detector works at runtime. It doesn't analyze our code statically! What does it mean? It means that we can have a potential race condition in our design that the race detector will not detect. For example:

```
package main

import "sync"

type Counter struct {
    sync.Mutex
    value int
}

func main() {
    counter := Counter{}

    for i := 0; i < 1; i++ {
        go func(i int) {
            counter.value++
        }(i)
    }
}
```

```
}
```

We will leave the code as shown in the preceding example. We will take all locks and unlocks from the code and launch a single Goroutine to update the `value` field:

```
$ go run -race main.go
$
```

No warnings, so the code is correct. Well, we know, by design, it's not. We can raise the number of Goroutines executed to two and see what happens:

```
for i := 0; i < 2; i++ {
    go func(i int) {
        counter.value++
    }(i)
}
```

Let's execute the program again:

```
$ go run -race main.go
WARNING: DATA RACE
Read at 0x00c42007a008 by goroutine 6:
  main.main.func1()
  [some_path]concurrency/race_detector/main.go:15 +0x44
Previous write at 0x00c42007a008 by goroutine 5:
  main.main.func1()
  [some_path]/concurrency/race_detector/main.go:15 +0x60
Goroutine 6 (running) created at:
  main.main()
  [some_path]/concurrency/race_detector/main.go:16 +0xad
Goroutine 5 (finished) created at:
  main.main()
  [some_path]/concurrency/race_detector/main.go:16 +0xad
=====
Found 1 data race(s)
exit status 66
```

Now yes, the race condition is detected. But what if we reduce the number of processors in use to just one? Will we have a race condition too?

```
$ GOMAXPROCS=1 go run -race main.go
$
```

It seems that no race condition has been detected. This is because the scheduler executed one Goroutine first and then the other, so, finally, the race condition didn't occur. But with a higher number of Goroutines it will also warn us about a race condition, even using only one core.

So, the race detector can help us to detect race conditions that are happening in our code, but it won't protect us from a bad design that is not immediately executing race conditions. A very useful feature that can save us from lots of headaches.

Channels

Channels are the second primitive in the language that allows us to write concurrent applications. We have talked a bit about channels in the *Communicating sequential processes* section.

Channels are the way we communicate between processes. We could be sharing a memory location and using mutexes to control the processes' access. But channels provide us with a more natural way to handle concurrent applications that also produces better concurrent designs in our programs.

Our first channel

Working with many Goroutines seems pretty difficult if we can't create some synchronization between them. The order of execution could be irrelevant as soon as they are synchronized. Channels are the second key feature to write concurrent applications in Go.

A TV channel in real life is something that connects an emission (from a studio) to millions of TVs (the receivers). Channels in Go work in a similar fashion. One or more Goroutines can work as emitters, and one or more Goroutine can act as receivers.

One more thing channels, by default, block the execution of Goroutines until something is received. It is as if our favourite TV show delays the emission until we turn the TV on so we don't miss anything.

How is this done in Go?

```
package main

import "fmt"

func main() {
    channel := make(chan string)
    go func() {
        channel <- "Hello World!"
    }()
}
```

```
    message := <-channel
    fmt.Println(message)
}
```

To create channels in Go, we use the same syntax that we use to create slices. The `make` keyword is used to create a channel, and we have to pass the keyword `chan` and the type that the channel will transport, in this case, strings. With this, we have a blocking channel with the name `channel`. Next, we launch a Goroutines that sends the message `Hello World!` to the channel. This is indicated by the intuitive arrow that shows the flow--the `Hello World!` text going to (`<-`) a channel. This works like an assignment in a variable, so we can only pass something to a channel by first writing the channel, then the arrow, and finally the value to pass. We cannot write "`Hello World!" -> channel`.

As we mentioned earlier, this channel is blocking the execution of Goroutines until a message is received. In this case, the execution of the `main` function is stopped until the message from the launched Goroutines reaches the other end of the channel in the line `message := <-channel`. In this case, the arrow points in the same direction, but it's placed before the channel, indicating that the data is being extracted from the channel and assigned to a new variable called `message` (using the new assignment "`:=`" operator).

In this case, we don't need to use a `WaitGroup` to synchronize the `main` function with the created Goroutines, as the default nature of channels is to block until data is received. But does it work the other way around? If there is no receiver when the Goroutine sends the message, does it continue? Let's edit this example to see this:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    channel := make(chan string)

    var waitGroup sync.WaitGroup

    waitGroup.Add(1)
    go func() {
        channel <- "Hello World!"
        println("Finishing goroutine")
        waitGroup.Done()
    }()
    time.Sleep(time.Second)
```

```
message := <-channel
fmt.Println(message)
waitGroup.Wait()
}
```

We are going to use the `Sleep` function again. In this case, we print a message when the Goroutine is finished. The big difference is in the `main` function. Now we wait one second before we listen to the channel for data:

```
$ go run main.go

Finishing goroutine
Hello World!
```

The output can differ because, again, there are no guarantees in the order of execution, but now we can see that no message is printed until one second has passed. After the initial delay, we start listening to the channel, take the data, and print it. So the emitter also has to wait for a cue from the other side of the channel to continue its execution.

To recap, channels are ways to communicate between Goroutines by sending data through one end and receiving it at the other (like a pipe). In their default state, an emitter Goroutine will block its execution until a receiver Goroutine takes the data. The same goes for a receiver Goroutine, which will block until some emitter sends data through the channel. So you can have passive listeners (waiting for data) or passive emitters (waiting for listeners).

Buffered channels

A buffered channel works in a similar way to default unbuffered channels. You also pass and take values from them by using the arrows, but, unlike unbuffered channels, senders don't need to wait until some Goroutine picks the data that they are sending:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    channel := make(chan string, 1)

    go func() {
        channel <- "Hello World!"
        println("Finishing goroutine")
    }()
}
```

```
    time.Sleep(time.Second)

    message := <-channel
    fmt.Println(message)
}
```

This example is like the first example we used for channels, but now we have set the capacity of the channel to one in the `make` statement. With this, we tell the compiler that this channel has a capacity of one string before getting blocked. So the first string doesn't block the emitter, but the second would. Let's run this example:

```
$ go run main.go

Finishing goroutine
Hello World!
```

Now we can run this small program as many times as we want--the output will always be in the same order. This time, we have launched the concurrent function and waited for one second. Previously, the anonymous function wouldn't continue until the second has passed and someone can pick the sent data. In this case, with a buffered channel, the data is held in the channel and frees the Goroutine to continue its execution. In this case, the Goroutine is always finishing before the wait time passes.

This new channel has a size of one, so a second message would block the Goroutine execution:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    channel := make(chan string, 1)

    go func() {
        channel <- "Hello World! 1"
        channel <- "Hello World! 2"
        println("Finishing goroutine")
    }()
}

time.Sleep(time.Second)

message := <-channel
fmt.Println(message)
}
```

Here, we add a second `Hello world! 2` message, and we provide it with an index. In this case, the output of this program could be like the following:

```
$ go run main.go
```

```
Hello World! 1
```

Indicating that we have just taken one message from the channel buffer, we have printed it, and the `main` function finished before the launched Goroutine could finish. The Goroutine got blocked when sending the second message and couldn't continue until the other end took the first message. Then it prints it so quickly that it doesn't have time to print the message to show the ending of the Goroutine. If you keep executing the program on the console, sooner or later the scheduler will finish the Goroutine execution before the main thread.

Directional channels

One cool feature about Go channels is that, when we use them as parameters, we can restrict their directionality so that they can be used only to send or to receive. The compiler will complain if a channel is used in the restricted direction. This feature applies a new level of static typing to Go apps and makes code more understandable and more readable.

We'll take a simple example with channels:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    channel := make(chan string, 1)

    go func(ch chan<- string) {
        ch <- "Hello World!"
        println("Finishing goroutine")
    }(channel)

    time.Sleep(time.Second)

    message := <-channel
    fmt.Println(message)
}
```

The line where we launch the new Goroutine `go func(ch chan<- string)` states that the channel passed to this function can only be used as an input channel, and you can't listen to it.

We can also pass a channel that will be used as a receiver channel only:

```
func receivingCh(ch <-chan string) {  
    msg := <-ch  
    println(msg)  
}
```

As you can see, the arrow is on the opposite side of the keyword `chan`, indicating an extracting operation from the channel. Keep in mind that the channel arrow always points left, to indicate a receiving channel, it must go on the left, and to indicate an inserting channel, it must go on the right.

If we try to send a value through this *receive only* channel, the compiler will complain about it:

```
func receivingCh(ch <-chan string) {  
    msg := <-ch  
    println(msg)  
    ch <- "hello"  
}
```

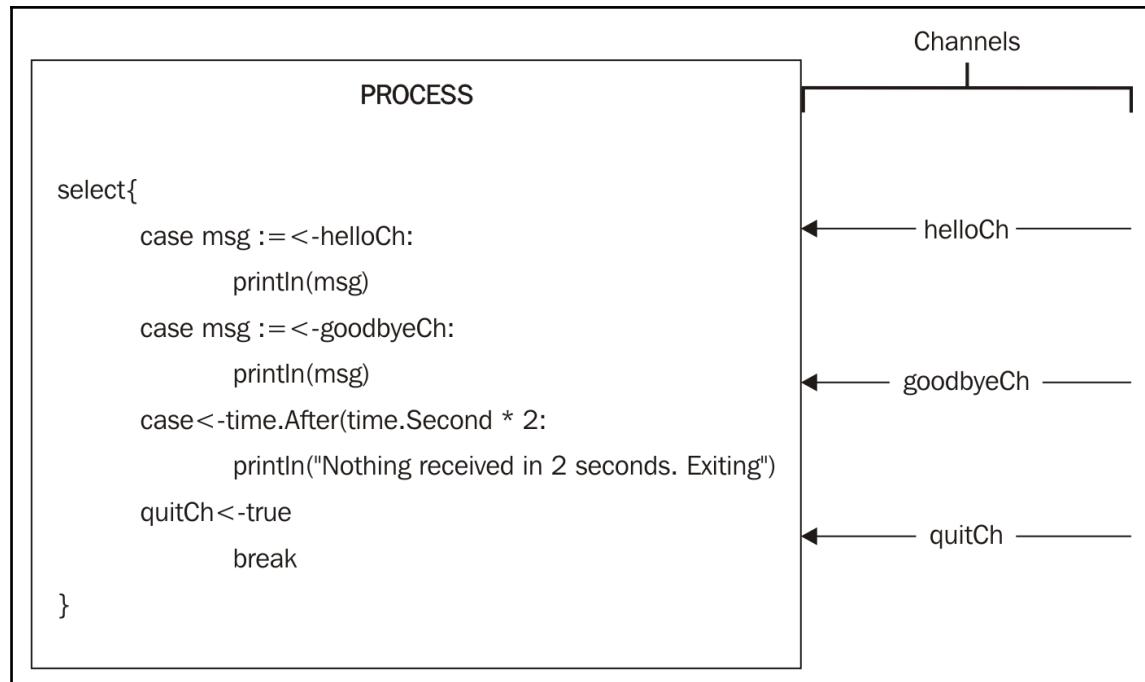
This function has a receive only channel that we will try to use to send the message `hello` through. Let's see what the compiler says:

```
$ go run main.go  
  
.main.go:20: invalid operation: ch <- "hello2" (send to receive-only type  
<-chan string)
```

It doesn't like it and asks us to correct it. Now the code is even more readable and safe, and we have just placed an arrow in front or behind the `chan` argument.

The select statement

The select statement is also a key feature in Go. It is used to handle more than one channel input within a Goroutine. In fact, it opens lots of possibilities, and we will use it extensively in the following chapters.



In the `select` structure, we ask the program to choose between one or more channels to receive their data. We can save this data in a variable and make something with it before finishing the `select`. The `select` structure is just executed once; it doesn't matter if it is listening to more channels, it will be executed only once and the code will continue executing. If we want it to handle the same channels more than once, we have to put it in a `for` loop.

We will make a small app that will send the message `hello` and the message `goodbye` to the same Goroutine, which will print them and exit if it doesn't receive anything else in five seconds.

First, we will make a generic function that sends a string over a channel:

```
func sendString(ch chan<- string, s string) {
    ch <- s
}
```

Now we can send a string over a channel by simply calling the `sendString` method. It's time for the receiver. The receiver will take messages from both channels--the one that sends `hello` messages and the one that sends `goodbye` messages. You can also see this in the previous diagram:

```
func receiver(helloCh, goodbyeCh <-chan string, quitCh chan<- bool) {
    for {
        select {
        case msg := <-helloCh:
            println(msg)
        case msg := <-goodbyeCh:
            println(msg)
        case <-time.After(time.Second * 2):
            println("Nothing received in 2 seconds. Exiting")
            quitCh <- true
            break
        }
    }
}
```

Let's start with the arguments. This function takes three channels--two receiving channels and one to send something through it. Then, it starts an infinite loop with the `for` keyword. This way we can keep listening to both channels forever.

Inside the scope of `select` block, we have to use a `case` for each channel we want to handle (have you realized how similar it is to the `switch` statement?). Let's see the three cases step by step:

- The first case takes the incoming data from the `helloCh` argument and saves it in a variable called `msg`. Then it prints the contents of this variable.
- The second case takes the incoming data from the `goodbyeCh` argument and saves it in a variable called `msg` too. Then it also prints the content of this variable.
- The third case is quite interesting. It calls the `time` function. After that, if we check its signature, it accepts a time and duration value and returns a receiving channel. This receiving channel will receive a time, the value of `time` after the specified duration has passed. In our example, we use the channel it returns as a timeout. Because the `select` is restarted after each handle, the timer is restarted too. This is a very simple way to set a timer to a Goroutine waiting for the response of one or many channels.

Everything is ready for the `main` function:

```
package main
```

```
import "time"

func main() {
    helloCh := make(chan string, 1)
    goodbyeCh := make(chan string, 1)
    quitCh := make(chan bool)
    go receiver(helloCh, goodbyeCh, quitCh)

    go sendString(helloCh, "hello!")

    time.Sleep(time.Second)

    go sendString(goodbyeCh, "goodbye!")
    <-quitCh
}
```

Again, step by step, we created the three channels that we'll need in this exercise. Then, we launched our `receiver` function in a different Goroutine. This Goroutine is handled by Go's scheduler and our program continues. We launched a new Goroutine to send the message `hello` to the `helloCh` arguments. Again, this is going to occur eventually when the Go's scheduler decides.

Our program continues again and waits for a second. In this break, Go's scheduler will have time to execute the `receiver` and the first message (if it hasn't done so yet), so the `hello!` message will appear on the console during the break.

A new message is sent over the `goodbye` channel with the `goodbye!` text in a new Goroutine, and our program continues again to a line where we wait for an incoming message in the `quitCh` argument.

We have launched three Goroutines already--the `receiver` that it is still running, the first message that had finished when the message was handled by the `select` statement, and the second message was been printed almost immediately and had finished too. So just the `receiver` is running at this moment, and if it doesn't receive any other message in the following two seconds, it will handle the incoming message from the `time` structure. After channel type, print a message to say that it is quitting, send a `true` to the `quitCh`, and break the infinite loop where it was looping.

Let's run this small app:

```
$ go run main.go

hello!
goodbye!
Nothing received in 2 seconds. Exiting
```

The result may not be very impressive, but the concept is clear. We can handle many incoming channels in the same Goroutine by using the select statement.

Ranging over channels too!

The last feature about channels that we will see is ranging over channels. We are talking about the range keyword. We have used it extensively to range over lists, and we can use it to range over a channel too:

```
package main

import "time"

func main() {
    ch := make(chan int)

    go func() {
        ch <- 1
        time.Sleep(time.Second)

        ch <- 2

        close(ch)
    }()
    for v := range ch {
        println(v)
    }
}
```

In this case, we have created an unbuffered channel, but it would work with a buffered one too. We launched a function in a new Goroutine that sends the number "1" over a channel, waits a second, sends the number "2", and closes the channel.

The last step is to range over the channel. The syntax is quite similar to a list range. We store the incoming data from the channel in the variable `v` and we print this variable to the console. The range keeps iterating until the channel is closed, taking data from the channel.

Can you guess the output of this little program?

```
$ go run main.go
1
2
```

Again, not very impressive. It prints the number "1", then waits a second, prints the number "2", and exits the application.

According to the design of this concurrent app, the range was iterates over possible incoming data from the

channel

until the concurrent Goroutine closes this channel. At that moment, the range finishes and the app can exit.

Range is very useful in taking data from a channel, and it's commonly used in fan-in patterns where many different Goroutines send data to the same channel.

Using it all - concurrent singleton

Now that we know how to create Goroutines and channels, we'll put all our knowledge in a single package. Think back to the first few chapter, when we explained the singleton pattern-it was some structure or variable that could only exist once in our code. All access to this structure should be done using the pattern described, but, in fact, it wasn't concurrent safe.

Now we will write with concurrency in mind. We will write a concurrent counter, like the one we wrote in the *mutexes* section, but this time we will solve it with channels.

Unit test

To restrict concurrent access to the `singleton` instance, just one Goroutine will be able to access it. We'll access it using channels--the first one to add one, the second one to get the current count, and the third one to stop the Goroutine.

We will add one 10,000 times using 10,000 different Goroutines launched from two different `singleton` instances. Then, we'll introduce a loop to check the count of the `singleton` until it is 5,000, but we'll write how much the count is before starting the loop.

Once the count has reached 5,000, the loop will exit and quit the running Goroutine-the test code looks like this:

```
package channel_singleton
import (
    "testing"
    "time"
```

```
"fmt"
)

func TestStartInstance(t *testing.T) {
    singleton := GetInstance()
    singleton2 := GetInstance()

    n := 5000

    for i := 0; i < n; i++ {
        go singleton.AddOne()
        go singleton2.AddOne()
    }

    fmt.Printf("Before loop, current count is %d\n", singleton.GetCount())

    var val int
    for val != n*2 {
        val = singleton.GetCount()
        time.Sleep(10 * time.Millisecond)
    }
    singleton.Stop()
}
```

Here, we can see the full test we'll use. After creating two instances of the `singleton`, we have created a `for` loop that launches the `AddOne` method 5,000 times from each instance. This is not happening yet; they are being scheduled and will be executed eventually. We are printing the count of the `singleton` instance to clearly see this eventuality; depending on the computer, it will print some number greater than 0 and lower than 10,000.

The last step before stopping the Goroutine that is holding the count is to enter a loop that checks the value of the count and waits 10 milliseconds if the value is not the expected value (10,000). Once it reaches this value, the loop will exit and we can stop the `singleton` instance.

We'll jump directly to the implementation as the requirement is quite simple.

Implementation

First of all, we'll create the Goroutine that will hold the count:

```
var addCh chan bool = make(chan bool)
var getCountCh chan chan int = make(chan chan int)
var quitCh chan bool = make(chan bool)
```

```
func init() {
    var count int

    go func(addCh <-chan bool, getCountCh <-chan chan int, quitCh <-chan
bool) {
        for {
            select {
            case <-addCh:
                count++
            case ch := <-getCountCh:
                ch <- count
            case <-quitCh:
                return
            }
        }
    } (addCh, getCountCh, quitCh)
}
```

We created three channels, as we mentioned earlier:

- The `addCh` channel is used to communicate with the action of adding one to the count, and receives a `bool` type just to signal "add one" (we don't need to send the number, although we could).
- The `getCountCh` channel will return a channel that will receive the current value of the count. Take a moment to reason about the `getCountCh` channel—it's a channel that receives a channel that receives integer types. It sounds a bit complicated, but it will make more sense when we finish the example, don't worry.
- The `quitCh` channel will communicate to the Goroutine that it should end its infinite loop and finish itself too.

Now we have the channels that we need to perform the actions we want. Next, we launch the Goroutine passing the channels as arguments. As you can see, we are restricting the direction of the channels to provide more type safety. Inside this Goroutine, we create an infinite `for` loop. This loop won't stop until a break is executed within it.

Finally, the `select` statement, if you remember, was a way to receive data from different channels at the same time. We have three cases, so we listen to the three incoming channels that entered as arguments:

- The `addCh` case will add one to the count. Remember that only one case can be executed on each iteration so that no Goroutine could be accessing the current count until we finish adding one.

- The `getCountCh` channel receives a channel that receives an integer, so we capture this new channel and send the current value through it to the other end.
- The `quitCh` channel breaks the `for` loop, so the Goroutine ends.

One last thing. The `init()` function in any package will get executed on program execution, so we don't need to worry about executing this function specifically from our code.

Now, we'll create the type that the tests are expecting. We will see that all the magic and logic is hidden from the end user in this type (as we have seen in the code of the test):

```
type singleton struct {}

var instance singleton
func GetInstance() *singleton {
    return &instance
}
```

The `singleton` type works similar to the way it worked in Chapter 2, *Creational Patterns - Singleton, Builder, Factory, Prototype, and Abstract Factory*, but this time it won't hold the count value. We created a local value for it called `instance`, and we return the pointer to this instance when we call the `GetInstance()` method. It is not strictly necessary to do it this way, but we don't need to allocate a new instance of the `singleton` type every time we want to access the count variable.

First, the `AddOne()` method will have to add one to the current count. How? By sending `true` to the `addCh` channel. That's simple:

```
func (s *singleton) AddOne() {
    addCh <- true
}
```

This small snippet will trigger the `addCh` case in our Goroutine in turn. The `addCh` case simply executes `count++` and finishes, letting `select` channel control flow that is executed on `init` function above to execute the next instruction:

```
func (s *singleton) GetCount() int {
    resCh := make(chan int)
    defer close(resCh)
    getCountCh <- resCh
    return <-resCh
}
```

The `GetCount` method creates a channel every time it's called and defers the action of closing it at the end of the function. This channel is unbuffered as we have seen previously in this chapter. An unbuffered channel blocks the execution until it receives some data. So we send this channel to `getCountCh` which is a channel too and, effectively, expects a `chan int` type to send the current count value back through it. The `GetCount()` method will not return until the value of `count` variable arrives to the `resCh` channel.

You might be thinking, why aren't we using the same channel in both directions to receive the value of the count? This way we will avoid an allocation. Well, if we use the same channel inside the `GetCount()` method, we will have two listeners in this channel--one in `select` statement, at the beginning of the file on the `init` function, and one there, so it could resolve to any of them when sending the value back:

```
func (s *singleton) Stop() {
    quitCh <- true
    close(addCh)
    close(getCountCh)
    close(quitCh)
}
```

Finally, we have to stop the Goroutine at some moment. The `Stop` method sends the value to the `singleton` type Goroutine so that the `quitCh` case is triggered and the `for` loop is broken. The next step is to close all channels so that no more data can be sent through them. This is very convenient when you know that you won't be using some of your channels anymore.

Time to execute the tests and take a look:

```
$ go test -v .

==== RUN TestStartInstance
Before loop, current count is 4911
--- PASS: TestStartInstance (0.03s)
PASS
ok
```

Very little code output, but everything has worked as expected. In the test, we printed the value of the count before entering the loop that iterates until it reaches the value 10,000. As we saw previously, the Go scheduler will try to run the content of the Goroutines using as many OS threads as you configured by using the `GOMAXPROCS` configuration. In my computer, it is set to 4 because my computer has four cores. But the point is that we can see that a lot of things can happen after launching a Goroutine (or 10,000) and the next execution line.

But what about its use of mutexes?

```
type singleton struct {
    count int
    sync.RWMutex
}

var instance singleton

func GetInstance() *singleton {
    return &instance
}

func (s *singleton) AddOne() {
    s.Lock()
    defer s.Unlock()
    s.count++
}

func (s *singleton) GetCount() int {
    s.RLock()
    defer s.RUnlock()
    return s.count
}
```

In this case, the code is much leaner. As we saw previously, we can embed the mutex within the `singleton` structure. The count is also held in the `count` field and the `AddOne()` and `GetCount()` methods lock and unlock the value to be concurrently safe.

One more thing. In this `singleton` instance, we are using the `RWMutex` type instead of the already known `sync.Mutex` type. The main difference here is that the `RWMutex` type has two types of locks--a read lock and a write lock. The read lock, executed by calling the `RLock` method, only waits if a write lock is currently active. At the same time, it only blocks a write lock, so that many read actions can be done in parallel. It makes a lot of sense; we don't want to block a Goroutine that wants to read a value just because another Goroutine is also reading the value—it won't change. The `sync.RWMutex` type helps us to achieve this logic in our code.

Summary

We have seen how to write a concurrent Singleton using mutexes and channels. While the channels example was more complex, it also shows the core power of Go's concurrency, as you can achieve complex levels of event-driven architectures by simply using channels.

Just keep in mind that, if you haven't written concurrent code in the past, it can take some time to start thinking concurrently in a comfortable way. But it's nothing that practice cannot solve.

We have seen the importance of designing concurrent apps to achieve parallelism in our programs. We have dealt with most of Go's primitives to write concurrent applications, and now we can write common concurrent design patterns.

9

Concurrency Patterns - Barrier, Future, and Pipeline Design Patterns

Now that we are familiar with the concepts of concurrency and parallelism, and we have understood how to achieve them by using Go's concurrency primitives, we can see some patterns regarding concurrent work and parallel execution. In this chapter we'll see the following patterns:

- Barrier is a very common pattern, especially when we have to wait for more than one response from different Goroutines before letting the program continue
- Future pattern allows us to write an algorithm that will be executed eventually in time (or not) by the same Goroutine or a different one
- Pipeline is a powerful pattern to build complex synchronous flows of Goroutines that are connected with each other according to some logic

Take a quick look at the description of the three patterns. They all describe some sort of logic to synchronize execution in time. It's very important to keep in mind that we are now developing concurrent structures with all the tools and patterns we have seen in the previous chapters. With Creational patterns we were dealing with creating objects. With the Structural patterns we were learning how to build idiomatic structures and in Behavioral patterns we were managing mostly with algorithms. Now, with Concurrency patterns, we will mostly manage the timing execution and order execution of applications that has more than one *flow*.

Barrier concurrency pattern

We are going to start with the Barrier pattern. Its purpose is simple--put up a barrier so that nobody passes until we have all the results we need, something quite common in concurrent applications.

Description

Imagine the situation where we have a microservices application where one service needs to compose its response by merging the responses of another three microservices. This is where the Barrier pattern can help us.

Our Barrier pattern could be a service that will block its response until it has been composed with the results returned by one or more different Goroutines (or services). And what kind of primitive do we have that has a blocking nature? Well, we can use a lock, but it's more idiomatic in Go to use an unbuffered channel.

Objectives

As its name implies, the Barrier pattern tries to stop an execution so it doesn't finish before it's ready to finish. The Barrier pattern's objectives are as follows:

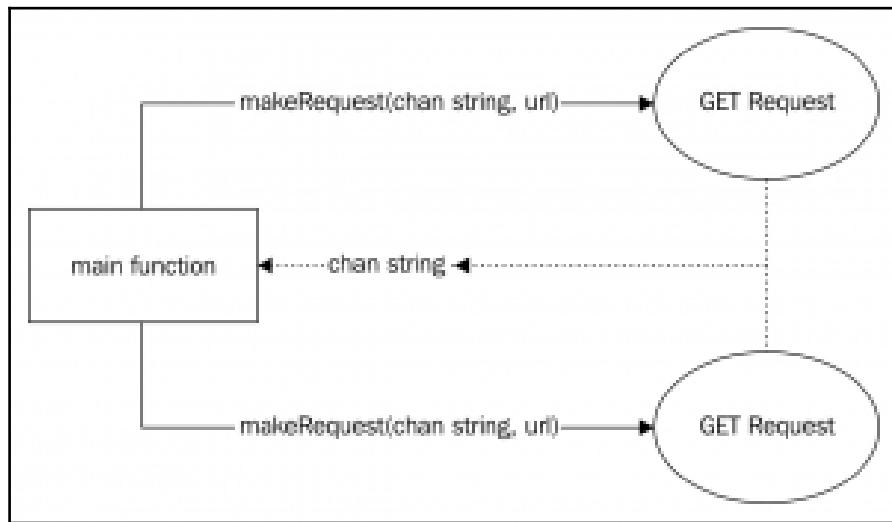
- Compose the value of a type with the data coming from one or more Goroutines.
- Control the correctness of any of those incoming data pipes so that no inconsistent data is returned. We don't want a partially filled result because one of the pipes has returned an error.

An HTTP GET aggregator

For our example, we are going to write a very typical situation in a microservices application—an app that performs two HTTP GET calls and joins them in a single response that will be printed on the console.

Our small app must perform each request in a different Goroutine and print the result on the console if both responses are correct. If any of them returns an error, then we print just the error.

The design must be concurrent, allowing us to take advantage of our multicore CPUs to make the calls in parallel:



In the preceding diagram, the solid lines represent calls and the dashed lines represent channels. The balloons are Goroutines, so we have two Goroutines launched by the `main` function (which could also be considered a Goroutine). These two functions will communicate back to the `main` function by using a **common channel** that they received when they were created on the `makeRequest` calls.

Acceptance criteria

Our main objective in this app is to get a merged response of two different calls, so we can describe our acceptance criteria like this:

- Print on the console the merged result of the two calls to <http://httpbin.org/headers> and <http://httpbin.org/User-Agent> URLs. These are a couple of public endpoints that respond with data from the incoming connections. They are very popular for testing purposes. You will need an internet connection to do this exercise.
- If any of the calls fails, it must not print any result-just the error message (or error messages if both calls failed).
- The output must be printed as a composed result when both calls have finished. It means that we cannot print the result of one call and then the other.

Unit test - integration

To write unit or integration tests for concurrent designs can sometimes be tricky, but this won't stop us from writing our awesome unit tests. We will have a single `barrier` method that accepts a set of endpoints defined as a `string` type. The barrier will make a `GET` request to each endpoint and compose the result before printing it out. In this case, we will write three integration tests to simplify our code so we don't need to generate mock responses:

```
package barrier

import (
    "bytes"
    "io"
    "os"
    "strings"
    "testing"
)

func TestBarrier(t *testing.T) {
    t.Run("Correct endpoints", func(t *testing.T) {
        endpoints := []string{"http://httpbin.org/headers",
        "http://httpbin.org/User-Agent"
        })
    })

    t.Run("One endpoint incorrect", func(t *testing.T) {
        endpoints := []string{"http://malformed-url",
        "http://httpbin.org/User-Agent"}
    })

    t.Run("Very short timeout", func(t *testing.T) {
        endpoints := []string{"http://httpbin.org/headers",
        "http://httpbin.org/User-Agent"}
    })
}
```

We have a single test that will execute three subtests:

- The first test makes two calls to the correct endpoints
- The second test will have an incorrect endpoint, so it must return an error
- The last test will return the maximum timeout time so that we can force a timeout error

We will have a function called `barrier` that will accept an undetermined number of endpoints in the form of strings. Its signature could be like this:

```
func barrier(endpoints ...string) {}
```

As you can see, the `barrier` function doesn't return any value because its result will be printed on the console. Previously, we have written an implementation of an `io.Writer` interface to emulate the writing on the operating system's `stdout` library. Just to change things a bit, we will capture the `stdout` library instead of emulating one. The process to capture the `stdout` library isn't difficult once you understand concurrency primitives in Go:

```
func captureBarrierOutput(endpoints ...string) string {
    reader, writer, _ := os.Pipe()

    os.Stdout = writer
    out := make(chan string)
    go func() {
        var buf bytes.Buffer
        io.Copy(&buf, reader)
        out <- buf.String()
    }()
    barrier(endpoints...)

    writer.Close()
    temp := <-out

    return temp
}
```

Don't feel daunted by this code; it's really simple. First we created a pipe; we have done this before in [Chapter 3, Structural Patterns - Adapter, Bridge, and Composite Design Patterns](#), when we talked about the Adapter design pattern. To recall, a pipe allows us to connect an `io.Writer` interface to an `io.Reader` interface so that the reader input is the Writer output. We define the `os.Stdout` as the writer. Then, to capture `stdout` output, we will need a different Goroutine that listens while we write to the console. As you know, if we write, we don't capture, and if we capture, we are not writing. The keyword here is `while`; it is a good rule of thumb that if you find this word in some definition, you'll probably need a concurrent structure. So we use the `go` keyword to launch a different Goroutine that copies reader input to a bytes buffer before sending the contents of the buffer through a channel (that we should have previously created).

At this point, we have a listening Goroutine, but we haven't printed anything yet, so we call our (not yet written) function `barrier` with the provided endpoints. Next, we have to close the writer to signal the Goroutine that no more input is going to come to it. Our channel called `out` blocks execution until some value is received (the one sent by our launched Goroutine). The last step is to return the contents captured from the console.

OK, so we have a function called `captureBarrierOutput` that will capture the outputs in `stdout` and return them as a string. We can write our tests now:

```
t.Run("Correct endpoints", func(t *testing.T) {
    endpoints := []string{"http://httpbin.org/headers",
    "http://httpbin.org/User-Agent"
    }

    result := captureBarrierOutput(endpoints...)
    if !strings.Contains(result, "Accept-Encoding") || strings.Contains
(result, "User-Agent")
    {
        t.Fail()
    }
    t.Log(result)
})
```

All the tests are very easy to implement. All in all, it is the `captureBarrierOutput` function that calls the `barrier` function. So we pass the endpoints and check the returned result. Our composed response directed to `http://httpbin.org` must contain the text `Accept-Encoding` and `User-Agent` in the responses of each endpoint. If we don't find those texts, the test will fail. For debugging purposes, we log the response in case we want to check it with the `-v` flag on the go test:

```
t.Run("One endpoint incorrect", func(t *testing.T) {
    endpoints := []string
    {
        "http://malformed-url", "http://httpbin.org/User-Agent"

    result := captureBarrierOutput(endpoints...)
    if !strings.Contains(result, "ERROR") {
        t.Fail()
    }
    t.Log(result)
})
```

This time we used an incorrect endpoint URL, so the response must return the error prefixed with the word `ERROR` that we will write ourselves in the `barrier` function.

The last function will reduce the timeout of the HTTP GET client to a minimum of 1 ms, so we force a timeout:

```
t.Run("Very short timeout", func(t *testing.T) {
    endpoints := []string {
        "http://httpbin.org/headers", "http://httpbin.org/User-Agent"
    }
    timeoutMilliseconds = 1
    result := captureBarrierOutput(endpoints...)
    if !strings.Contains(result, "Timeout") {
        t.Fail()
    }
    t.Log(result)
})
```

The `timeoutMilliseconds` variable will be a package variable that we will have to define later during implementation.

Implementation

We needed to define a package variable called `timeoutMilliseconds`. Let's start from there:

```
package barrier

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "time"
)

var timeoutMilliseconds int = 5000
```

The initial timeout delay is 5 seconds (5,000 milliseconds) and we will need those packages in our code.

OK, so we need a function that launches a Goroutine for each endpoint URL. Do you remember how we achieve the communication between Goroutines? Exactly--channels! So we will need a channel to handle responses and a channel to handle errors.

But we can simplify it a bit more. We will receive two correct responses, two errors, or a response and an error; in any case, there are always two responses, so we can join errors and responses in a merged type:

```
type barrierResp struct {
    Err  error
    Resp string
}
```

So, each Goroutine will send back a value of the `barrierResp` type. This value will have a value for `Err` or a value for the `Resp` field.

The procedure is simple: we create a channel of size 2, the one that will receive responses of the `barrierResp` type, we launch both requests and wait for two responses, and then check to see if there is any error:

```
func barrier(endpoints ...string) {
    requestNumber := len(endpoints)

    in := make(chan barrierResp, requestNumber)
    defer close(in)

    responses := make([]barrierResp, requestNumber)

    for _, endpoint := range endpoints {
        go makeRequest(in, endpoint)
    }

    var hasError bool
    for i := 0; i < requestNumber; i++ {
        resp := <-in
        if resp.Err != nil {
            fmt.Println("ERROR: ", resp.Err)
            hasError = true
        }
        responses[i] = resp
    }

    if !hasError {
        for _, resp := range responses {
            fmt.Println(resp.Resp)
        }
    }
}
```

Following the previous description, we created a buffered channel called `in`, making it the size of the incoming endpoints, and we deferred channel closing. Then, we launched a function called `makeRequest` with each endpoint and the response channel.

Now we will loop twice, once for each endpoint. In the loop, we block the execution waiting for data from the `in` channel. If we find an error, we print it prefixed with the word `ERROR` as we expect in our tests, and set `hasErrorvar` to true. After two responses, if we don't find any error (`hasError== false`) we print every response and the channel will be closed.

We still lack the `makeRequest` function:

```
func makeRequest(out chan<- barrierResp, url string) {
    res := barrierResp{}
    client := http.Client{
        Timeout: time.Duration(timeoutMilliseconds) * time.Millisecond,
    }

    resp, err := client.Get(url)
    if err != nil {
        res.Err = err
        out <- res
        return
    }

    byt, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        res.Err = err
        out <- res
        return
    }

    res.Resp = string(byt)
    out <- res
}
```

The `makeRequest` function is a very straightforward functions that accepts a channel to output `barrierResp` values to and a URL to request. We create an `http.Client` and set its `timeout` field to the value of the `timeoutMilliseconds` package variable. This is how we can change the timeout delay before the `in` function tests. Then, we simply make the GET call, take the response, parse it to a byte slice, and send it through the `out` channel.

We do all this by filling a variable called `res` of the `barrierResp` type. If we find an error while performing a GET request or parsing the body of the result, we fill the `res.Err` field, send it to the `out` channel (which has the opposite side connected to the original Goroutine), and exit the function (so we don't send two values through the `out` channel by mistake).

Time to run the tests. Remember that you need an Internet connection, or the first two tests will fail. We will first try the test that has two endpoints that are correct:

```
go test -run=TestBarrier/Correct_endpoints -v .
==== RUN TestBarrier
==== RUN TestBarrier/Correct_endpoints
--- PASS: TestBarrier (0.54s)
    --- PASS: TestBarrier/Correct_endpoints (0.54s)
        barrier_test.go:20: {
            "headers": {
                "Accept-Encoding": "gzip", "Host": "httpbin.org", "User-Agent": "Go-http-client/1.1"
            }
        }
    {
        "User-Agent": "Go-http-client/1.1"
    }
ok
```

Perfect. We have a JSON response with a key, `headers`, and another JSON response with a key `User-Agent`. In our integration tests, we were looking for the strings, `User-Agent` and `Accept-Encoding`, which are present, so the test has passed successfully.

Now we will run the test that has an incorrect endpoint:

```
go test -run=TestBarrier/One_endpoint_incorrect -v .
==== RUN TestBarrier
==== RUN TestBarrier/One_endpoint_incorrect
--- PASS: TestBarrier (0.27s)
    --- PASS: TestBarrier/One_endpoint_incorrect (0.27s)
        barrier_test.go:31: ERROR: Get http://malformed-url: dial tcp:
lookup malformed-url: no such host
ok
```

We can see that we have had an error where `http://malformed-url` has returned a *no such host* error. A request to this URL must return a text with the word `ERROR:` prefixed, as we stated during the acceptance criteria, that's why this test is correct (we don't have a false positive).



In testing, it's very important to understand the concepts of "false positive" and "false negative" tests. A false positive test is roughly described as a test that passes a condition when it shouldn't (result: all passed) while the false negative is just the reverse (result: test failed). For example, we could be testing that a string is returned when doing the requests but, the returned string could be completely empty! This will lead to a false negative, a test that doesn't fail even when we are checking a behavior that is incorrect on purpose (a request to `http://malformed-url`).

The last test reduced the timeout time to 1 ms:

```
go test -run=TestBarrier/Very_short_timeout -v .
==== RUN TestBarrier
==== RUN TestBarrier/Very_short_timeout
--- PASS: TestBarrier (0.00s)
--- PASS: TestBarrier/Very_short_timeout (0.00s)
    barrier_test.go:43: ERROR: Get http://httpbin.org/User-Agent:
    net/http: request canceled while waiting for connection (Client.Timeout
    exceeded while awaiting headers)
    ERROR: Get http://httpbin.org/headers: net/http: request canceled
    while waiting for connection (Client.Timeout exceeded while awaiting
    headers)
ok
```

Again, the test passed successfully and we have got two timeout errors. The URLs were correct, but we didn't have a response in less than one millisecond, so the client has returned a timeout error.

Waiting for responses with the Barrier design pattern

The Barrier pattern opens the door of microservices programming with its composable nature. It could be considered a Structural pattern, as you can imagine.

The Barrier pattern is not only useful to make network requests; we could also use it to split some task into multiple Goroutines. For example, an expensive operation could be split into a few smaller operations distributed in different Goroutines to maximize parallelism and achieve better performance.

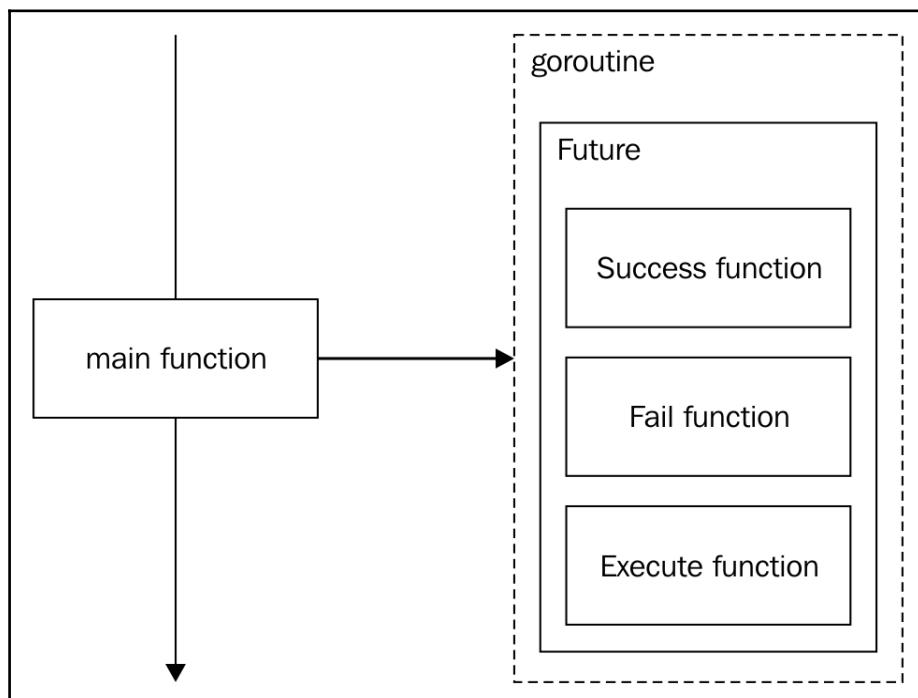
Future design pattern

The Future design pattern (also called **Promise**) is a quick and easy way to achieve concurrent structures for asynchronous programming. We will take advantage of first class functions in Go to develop *Futures*.

Description

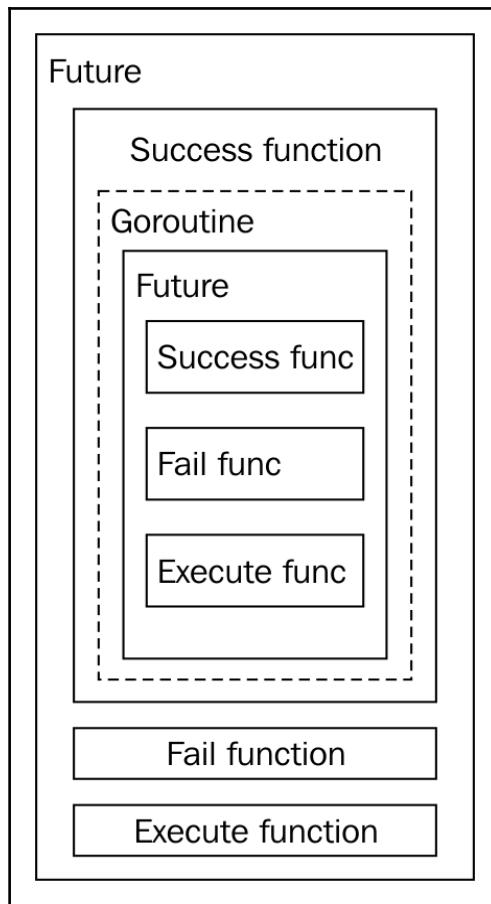
In short, we will define each possible behavior of an action before executing them in different Goroutines. Node.js uses this approach, providing event-driven programming by default. The idea here is to achieve a *fire-and-forget* that handles all possible results in an action.

To understand it better, we can talk about a type that has embedded the behavior in case an execution goes well or in case it fails.



In the preceding diagram, the `main` function launches a **Future** within a new Goroutine. It won't wait for anything, nor will it receive any progress of the Future. It really fires and forgets it.

The interesting thing here is that we can launch a new Future within a Future and embed as many Futures as we want in the same Goroutine (or new ones). The idea is to take advantage of the result of one Future to launch the next. For example:



Here, we have the same Future. In this case, if the `Execute` function returned a correct result, the `Success` function is executed, and only in this case we execute a new Goroutine with another Future inside (or even without a Goroutine).

This is a kind of lazy programming, where a Future could be calling to itself indefinitely or just until some rule is satisfied. The idea is to define the behavior in advance and let the future resolve the possible solutions.

Objectives

With the Future pattern, we can launch many new Goroutines, each with an action and its own handlers. This enables us to do the following:

- Delegate the action handler to a different Goroutine
- Stack many asynchronous calls between them (an asynchronous call that calls another asynchronous call in its results)

A simple asynchronous requester

We are going to develop a very simple example to try to understand how a Future works. In this example, we will have a method that returns a string or an error, but we want to execute it concurrently. We have learned ways to do this already. Using a channel, we can launch a new Goroutine and handle the incoming result from the channel.

But in this case, we will have to handle the result (string or error), and we don't want this. Instead, we will define what to do in case of success and what to do in case of error and fire-and-forget the Goroutine.

Acceptance criteria

We don't have functional requirements for this task. Instead, we will have technical requirements for it:

- Delegate the function execution to a different Goroutine
- The function will return a string (maybe) or an error
- The handlers must be already defined before executing the function
- The design must be reusable

Unit tests

So, as we mentioned, we will use first class functions to achieve this behavior, and we will need three specific types of function:

- type SuccessFunc func(string): The SuccessFunc function will be executed if everything went well. Its string argument will be the result of the operation, so this function will be called by our Goroutine.
- type FailFunc func(error): The FailFunc function handles the opposite result, that is, when something goes wrong, and, as you can see, it will return an error.
- type ExecuteStringFunc func() (string, error): Finally, the ExecuteStringFunc function is a type that defines the operation we want to perform. Maybe it will return a string or an error. Don't worry if this all seems confusing; it will be clearer later.

So, we create the `future` object, we define a success behavior, we define a fail behavior, and we pass an `ExecuteStringFunc` type to be executed. In the implementation file, we'll need a new type:

```
type MaybeString struct { }
```

We will also create two tests in the `_test.go` file:

```
package future

import (
    "errors"
    "testing"
    "sync"
)

func TestStringOrError_Execute(t *testing.T) {
    future := &MaybeString{}
    t.Run("Success result", func(t *testing.T) {
        ...
    })
    t.Run("Error result", func(t *testing.T) {
        ...
    })
}
```

We will define functions by chaining them, as you would usually see in Node.js. Code like this is compact and not particularly difficult to follow:

```
t.Run("Success result", func(t *testing.T) {
    future.Success(func(s string) {
        t.Log(s)
    }).Fail(func(e error) {
        t.Fail()
    })
    future.Execute(func() (string, error) {
        return "Hello World!", nil
    })
})
```

The `future.Success` function must be defined in the `MaybeString` structure to accept a `SuccessFunc` function that will be executed if everything goes correctly and return the same pointer to the `future` object (so we can keep chaining). The `Fail` function must also be defined in the `MaybeString` structure and must accept a `FailFunc` function to later return the pointer. We return the pointer in both cases so we can define the `Fail` and the `Success` or vice versa.

Finally, we use the `Execute` method to pass an `ExecuteStringFunc` type (a function that accepts nothing and returns a string or an error). In this case, we return a string and nil, so we expect that the `SuccessFunc` function will be executed and we log the result to the console. In case that fail function is executed, the test has failed because the `FailFunc` function shouldn't be executed for a returned nil error.

But we still lack something here. We said that the function must be executed asynchronously in a different Goroutine, so we have to synchronize this test somehow so that it doesn't finish too soon. Again, we can use a channel or a `sync.WaitGroup`:

```
t.Run("Success result", func(t *testing.T) {
    var wg sync.WaitGroup
    wg.Add(1)
    future.Success(func(s string) {
        t.Log(s)

        wg.Done()
    }).Fail(func(e error) {
        t.Fail()
        wg.Done()
    })
    future.Execute(func() (string, error) {
        return "Hello World!", nil
    })
    wg.Wait()
})
```

```
})
```

We have seen WaitGroups before in the previous channel. This WaitGroup is configured to wait for one signal (`wg.Add(1)`). The `Success` and `Fail` methods will trigger the `Done()` method of the WaitGroup to allow execution to continue and finish testing (that is why the `Wait()` method is at the end). Remember that each `Done()` method will subtract one from the WaitGroup, and we have added only one, so our `Wait()` method will only block until one `Done()` method is executed.

Using what we know of making a `Success` result unit test, it's easy to make a Failed result unit test by swapping the `t.Fail()` method call from the error to success so that the test fails if a call to success is done:

```
t.Run("Failed result", func(t *testing.T) {
    var wg sync.WaitGroup
    wg.Add(1)
    future.Success(func(s string) {
        t.Fail()
        wg.Done()
    }).Fail(func(e error) {
        t.Log(e.Error())
        wg.Done()
    })
    future.Execute(func() (string, error) {
        return "", errors.New("Error occurred")
    })
    wg.Wait()
})
```

If you are using an IDE like me, your `Success`, `Fail`, and `Execute` method calls must be in red. This is because we lack our method's declaration in the implementation file:

```
package future

type SuccessFunc func(string)
type FailFunc func(error)
type ExecuteStringFunc func() (string, error)

type MaybeString struct {
    ...
}

func (s *MaybeString) Success(f SuccessFunc) *MaybeString {
    return nil
}

func (s *MaybeString) Fail(f FailFunc) *MaybeString {
```

```
    return nil
}

func (s *MaybeString) Execute(f ExecuteStringFunc) {
    ...
}
```

Our test seems ready to execute. Let's try it out:

```
go test -v .
== RUN TestStringOrError_Execute
== RUN TestStringOrError_Execute/Success_result
fatal error: all goroutines are asleep - deadlock!
goroutine 1 [chan receive]:
testing.(*T).Run(0xc4200780c0, 0x5122e9, 0x19, 0x51d750, 0xc420041d30)
    /usr/lib/go/src/testing/testing.go:647 +0x316
testing.RunTests.func1(0xc4200780c0)
    /usr/lib/go/src/testing/testing.go:793 +0x6d
testing.tRunner(0xc4200780c0, 0xc420041e20)
    /usr/lib/go/src/testing/testing.go:610 +0x81
testing.RunTests(0x51d758, 0x5931e0, 0x1, 0x1, 0x50feb4)
    /usr/lib/go/src/testing/testing.go:799 +0x2f5
testing.(*M).Run(0xc420041ee8, 0xc420014550)
    /usr/lib/go/src/testing/testing.go:743 +0x85
main.main()
    go-design-patterns/future/_test/_testmain.go:54 +0xc6
...continue
```

Well... the tests have failed, yes... but not in a controllable way. Why is this? We don't have any implementation yet, so no Success or Fail functions are being executed either. Our WaitGroup is waiting forever for a call to the Done () method that will never arrive, so it can't continue and finish the test. That's the meaning of *All Goroutines are asleep - deadlock!*. In our specific example, it would mean *Nobody is going to call Done(), so we are dead!*.



Thanks to the Go compiler and the runtime executor, we can detect deadlocks easily. Imagine if Go runtime couldn't detect deadlocks--we would be effectively stuck in a blank screen without knowing what was wrong.

So how can we solve this? Well, an easy way would be with a timeout that calls the Done () method after waiting a while for completion. For this code, it's safe to wait for 1 second because it's not doing long-running operations.

We will declare a `timeout` function within our `test` file that waits for a second, then prints a message, sets the test as failed, and lets the `WaitGroup` continue by calling its `Done()` method:

```
func timeout(t *testing.T, wg *sync.WaitGroup) {
    time.Sleep(time.Second)
    t.Log("Timeout!")

    t.Fail()
    wg.Done()
}
```

The final look of each subtest is similar to our previous example of the "Success result":

```
t.Run("Success result", func(t *testing.T) {
    var wg sync.WaitGroup
    wg.Add(1)

    //Timeout!
    go timeout(t, wg)
    // ...
})
```

Let's see what happens when we execute our tests again:

```
go test -v .
==== RUN    TestStringOrError_Execute
==== RUN    TestStringOrError_Execute/Success_result
==== RUN    TestStringOrError_Execute/Failed_result
---- FAIL: TestStringOrError_Execute (2.00s)
      --- FAIL: TestStringOrError_Execute/Success_result (1.00s)
          future_test.go:64: Timeout!
      --- FAIL: TestStringOrError_Execute/Failed_result (1.00s)
          future_test.go:64: Timeout!
FAIL
exit status 1
FAIL
```

Our tests failed, but in a controlled way. Look at the end of the `FAIL` lines--notice how the elapsed time is 1 second because it has been triggered by the timeout, as we can see in the logging messages.

It's time to pass to the implementation.

Implementation

According to our tests, the implementation must take a `SuccessFunc`, a `FailFunc`, and an `ExecuteStringFunc` function in a chained fashion within the `MaybeString` type and launches the `ExecuteStringFunc` function asynchronously to call `SuccessFunc` or `FailFunc` functions according to the returned result of the `ExecuteStringFunc` function.

The chain is implemented by storing the functions within the type and returning the pointer to the type. We are talking about our previously declared type methods, of course:

```
type MaybeString struct {
    successFunc SuccessFunc
    failFunc    FailFunc
}

func (s *MaybeString) Success(f SuccessFunc) *MaybeString {
    s.successFunc = f
    return s
}

func (s *MaybeString) Fail(f FailFunc) *MaybeString {
    s.failFunc = f
    return s
}
```

We needed two fields to store the `SuccessFunc` and `FailFunc` functions, which are named the `successFunc` and `failFunc` fields respectively. This way, calls to the `Success` and `Fail` methods simply store their incoming functions to our new fields. They are simply setters that also return the pointer to the specific `MaybeString` value. These type methods take a pointer to the `MaybeString` structure, so don't forget to put "*" on `MaybeString` after the `func` declaration.

`Execute` takes the `ExecuteStringFunc` method and executes it asynchronously. This seems quite simple with a Goroutine, right?

```
func (s *MaybeString) Execute(f ExecuteStringFunc) {
    go func(s *MaybeString) {
        str, err := f()
        if err != nil {
            s.failFunc(err)
        } else {
            s.successFunc(str)
        }
    }(s)
```

```
}
```

Looks quite simple because it is simple! We launch the Goroutine that executes the `f` method (an `ExecuteStringFunc`) and takes its result--maybe a string and maybe an error. If an error is present, we call the field `failFunc` in our `MaybeString` structure. If no error is present, we call the `successFunc` field. We use a Goroutine to delegate a function execution and error handling so our Goroutine doesn't have to do it.

Let's run unit tests now:

```
go test -v .
==== RUN    TestStringOrError_Execute
==== RUN    TestStringOrError_Execute/Success_result
==== RUN    TestStringOrError_Execute/Failed_result
--- PASS: TestStringOrError_Execute (0.00s)
    --- PASS: TestStringOrError_Execute/Success_result (0.00s)
        future_test.go:21: Hello World!
    --- PASS: TestStringOrError_Execute/Failed_result (0.00s)
        future_test.go:49: Error occurred
PASS
ok
```

Great! Look how the execution time is now nearly zero, so our timeouts have not been executed (actually, they were executed, but the tests already finished and their result was already stated).

What's more, now we can use our `MaybeString` type to asynchronously execute any type of function that accepts nothing and returns a string or an error. A function that accepts nothing seems a bit useless, right? But we can use closures to introduce a context into this type of function.

Let's write a `setContext` function that takes a string as an argument and returns an `ExecuteStringFunc` method that returns the previous argument with the suffix `Closure!`:

```
func setContext(msg string) ExecuteStringFunc {
    msg = fmt.Sprintf("%d Closure!\n", msg)
    return func() (string, error) {
        return msg, nil
    }
}
```

So, we can write a new test that uses this closure:

```
t.Run("Closure Success result", func(t *testing.T) {
    var wg sync.WaitGroup
```

```
wg.Add(1)
//Timeout!
go timeout(t, &wg)

future.Success(func(s string) {
    t.Log(s)
    wg.Done()
}).Fail(func(e error) {
    t.Fail()
    wg.Done()
})
future.Execute(setContext("Hello"))
wg.Wait()
})
```

The `setContext` function returns an `ExecuteStringFunc` method it can pass directly to the `Execute` function. We call the `setContext` function with an arbitrary text that we know will be returned.

Let's execute our tests again. Now everything has to go well!

```
go test -v .
==== RUN TestStringOrError_Execute
==== RUN TestStringOrError_Execute/Success_result
==== RUN TestStringOrError_Execute/Failed_result
==== RUN TestStringOrError_Execute/Closure_Success_result
--- PASS: TestStringOrError_Execute (0.00s)
    --- PASS: TestStringOrError_Execute/Success_result (0.00s)
        future_test.go:21: Hello World!
    --- PASS: TestStringOrError_Execute/Failed_result (0.00s)
        future_test.go:49: Error occurred
    --- PASS: TestStringOrError_Execute/Closure_Success_result (0.00s)
        future_test.go:69: Hello Closure!
PASS
ok
```

It gave us an OK too. Closure test shows the behavior that we explained before. By taking a message "Hello" and appending it with something else ("Closure!"), we can change the context of the text we want to return. Now scale this to a HTTP GET call, a call to a database, or anything you can imagine. It will just need to end by returning a string or an error. Remember, however, that everything within the `setContext` function but outside of the anonymous function that we are returning is not concurrent, and will be executed asynchronously before calling `execute`, so we must try to put as much logic as possible within the anonymous function.

Putting the Future together

We have seen a good way to achieve asynchronous programming by using a function type system. However, we could have done it without functions by setting an interface with `Success`, `Fail`, and `Execute` methods and the types that satisfy them, and using the Template pattern to execute them asynchronously, as we have previously seen in this chapter. It is up to you!

Pipeline design pattern

The third and final pattern we will see in this chapter is the Pipeline pattern. You will use this pattern heavily in your concurrent structures, and we can consider it one of the most useful too.

Description

We already know what a pipeline is. Every time that we write any function that performs some logic, we are writing a pipeline: If *this* then *that*, or else *something else*. Pipelines pattern can be made more complex by using a few functions that call to each other. They can even get looped in their out execution.

The Pipeline pattern in Go works in a similar fashion, but each step in the Pipeline will be in a different Goroutine and communication, and synchronizing will be done using channels.

Objectives

When creating a Pipeline, we are mainly looking for the following benefits:

- We can create a concurrent structure of a multistep algorithm
- We can exploit the parallelism of multicore machines by decomposing an algorithm in different Goroutines

However, just because we decompose an algorithm in different Goroutines doesn't necessarily mean that it will execute the fastest. We are constantly talking about CPUs, so ideally the algorithm must be CPU-intensive to take advantage of a concurrent structure. The overhead of creating Goroutines and channels could make an algorithm smaller.

A concurrent multi-operation

We are going to do some math for our example. We are going to generate a list of numbers starting with 1 and ending at some arbitrary number N. Then we will take each number, power it to 2, and sum the resulting numbers to a unique result. So, if $N=3$, our list will be [1,2,3]. After powering them to 2, our list becomes [1,4,9]. If we sum the resulting list, the resulting value is 14.

Acceptance criteria

Functionally speaking, our Pipeline pattern needs to raise to the power of 2 every number and then sum them all. It will be divided into a number generator and two operations, so:

1. Generate a list from 1 to N where N can be any integer number.
2. Take each number of this generated list and raise it to the power of 2.
3. Sum each resulting number into a final result and return it.

Beginning with tests

We will create only one function that will manage everything. We will call this function `LaunchPipeline` to simplify things. It will take an integer as an argument, which will be our N number, the number of items in our list. The declaration in the implementation file looks like this:

```
package pipelines

func LaunchPipeline(amount int) int {
    return 0
}
```

In our test file, we will create a table of tests by using a slice of slices:

```
package pipelines

import "testing"

func TestLaunchPipeline(t *testing.T) {
    tableTest := [][]int{
        {3, 14},
        {5, 55},
    }
    // ...
}
```

Our table is a slice of slices of integer types. On each slice, the first integer represents the list size and the second position represents the item within the list. It is, effectively, a matrix. When passing 3, it must return 14. When passing 5, it must return 55. Then we have to iterate over the table and pass the first index of each array to the `LaunchPipeline` function:

```
// ...

var res int
for _, test := range tableTest {
    res = LaunchPipeline(test[0])
    if res != test[1] {
        t.Fatal()
    }

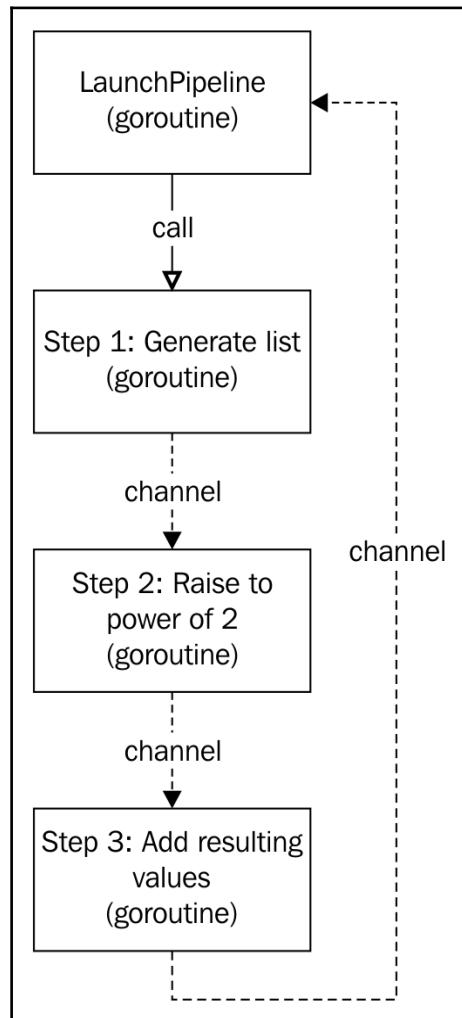
    t.Logf("%d == %d\n", res, test[1])
}
}
```

Using `range`, we get every row in the matrix. Each row is contained in a temporary variable called `test`. `test[0]` represents N and `test[1]` the expected result. We compare the expected result with the returning value of the `LaunchPipeline` function. If they aren't the same, the test fails:

```
go test -v .
===[  
--- RUN TestLaunchPipeline  
--- FAIL: TestLaunchPipeline (0.00s)  
    pipeline_test.go:15:  
FAIL  
exit status 1  
FAIL
```

Implementation

The key for our implementation is to separate every operation in a different Goroutine and connect them with channels. The `LaunchPipeline` function is the one that orchestrates them all, as shown in the following diagram:



The operation consists of three steps: generate a list of numbers, raise them to the power of 2, and add the resulting numbers.

Each step in this Pipeline pattern will have the following structure:

```
func functionName(in <-chan int) (<-chan int) {
    out := make(chan bool, 100)

    go func() {
        for v := range in {
            // Do something with v and send it to channel out
        }
    }

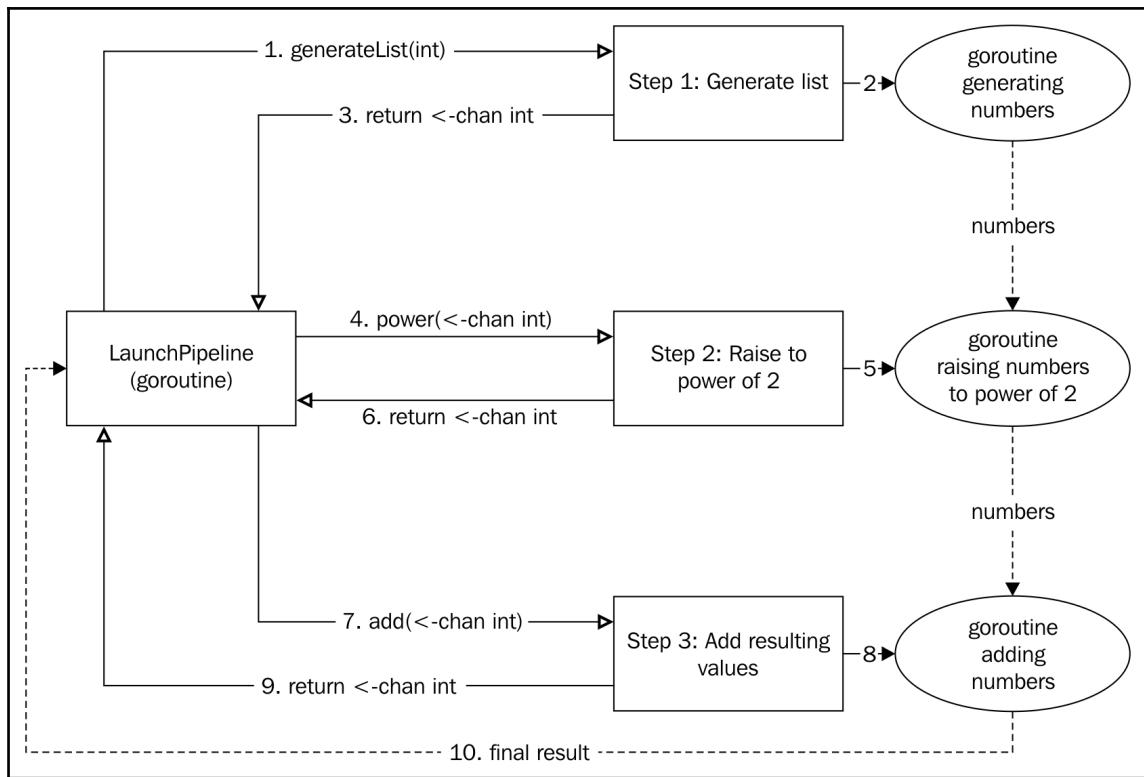
    close(out)
    }()

    return out
}
```

This function represents a common step. Let's dissect it in the same order that the Go scheduler will probably take to execute it:

1. The `functionName` function will commonly receive a channel to take values from (`in <-chan int`). We call it the `in` function, as in the word incoming. We can't send values through it within the scope of this function; that's why the arrow points `out` of the keyword `chan`.
2. The `functionName` function returns a channel (`<-chan in`) that the function caller will only be allowed to take values from (again, represented by the arrow pointing `out` of the keyword `chan`). This also means that any value that goes through that channel must be generated within the scope of the function.
3. In the first line of the function, we create a channel called `out` that will be the return of the function (*point 2* in this list).
4. Then, we will launch a new Goroutine. Its scope will enter into play after returning this function, so let's continue.
5. We return the previously created `out` channel.
6. Eventually, after finishing the execution of the function and returning the channel `out`, the Goroutine executes. It will take values from the `in` channel until it's closed. So the caller of this function is responsible for closing this channel, otherwise the Goroutine will never end!
7. When the `in` channel is closed, the `for` loop finishes and we close the `out` channel. Any Goroutine making use of this channel will not receive any new values since the last that was sent.

The only step that doesn't completely fit this approach is the first step that receives a number, representing the upper threshold on the list instead of a channel of incoming values. So, if we code this operation for each step in our pipeline, the final diagram looks more like this:



Although the idea is exactly the same, now we can see that it's the function `LaunchPipeline` that is the one that is going to be receiving channels and sending them back to the next step in the Pipeline. Using this diagram, we can clearly see the flow of the pipeline creation by following the numbers of the arrows. A solid arrow represents a function call and a dashed arrow a channel.

Let's look a little more closely at the code.

The list generator

The first step in the operation is list generation. The list starts at 1 and we will receive an integer representing the higher threshold. We have to pass each number in the list to the next step:

```
func generator(max int) <-chan int {
    outChInt := make(chan int, 100)

    go func() {
        for i := 1; i <= max; i++ {
            outChInt <- i
        }

        close(outChInt)
    }()
    return outChInt
}
```

As we mentioned earlier, this is the pattern that we will follow in each step: create a channel, launch the Goroutine that will send the data through the channel, and immediately return the channel. This Goroutine will iterate from 1 to the max argument, which is the higher threshold for our list, and send each number through the channel. After sending every number, the channel is closed so that no more data can be sent through it, but the data already buffered can be retrieved.

Raising numbers to the power of 2

The second step will take every incoming number from the first step's channel (that is taken from the arguments) and raise it to the power of 2. Every result must be sent to the third step using a new channel:

```
func power(in <-chan int) <-chan int {
    out := make(chan int, 100)

    go func() {
        for v := range in {
            out <- v * v
        }
        close(out)
    }()
    return out
}
```

We use the same pattern again: create a channel and launch the Goroutine while we return the created channel.



The `for-range` loop keeps taking values from a channel indefinitely until the channel is closed.

Final reduce operation

The third and final step receives every number from the second step and keeps adding them to a local value until the connection channel is closed:

```
func sum(in <-chan int) <-chan int {
    out := make(chan int, 100)
    go func() {
        var sum int

        for v := range in {
            sum += v
        }

        out <- sum
        close(out)
    }()
}

return out
}
```

The function `sum` also takes a channel as an argument (the one returned from *step 2*). It also follows the same pattern of creating a channel, launching the Goroutine, and returning a channel. Goroutine keeps adding values to a variable called `sum` until the `in` channel is closed. When the `in` channel is closed, the value of `sum` is sent to the `out` channel, and it's immediately closed.

Launching the Pipeline pattern

Finally, we can implement the `LaunchPipeline` function:

```
func LaunchPipeline(amount int) int {
    firstCh := generator(amount)
    secondCh := power(firstCh)
    thirdCh := sum(secondCh)
```

```
    result := <-thirdCh

    return result
}
```

The function generator first returns the channel that is passed to the power function. The power function returns the second channel that is passed to the sum function. The function sum finally returns the first channel that will receive a unique value, the result. Let's try to test this now:

```
go test -v .
==== RUN    TestLaunchPipeline
--- PASS: TestLaunchPipeline (0.00s)
    pipeline_test.go:18: 14 == 14
    pipeline_test.go:18: 55 == 55
PASS
ok
```

Awesome! It's worth mentioning that the LaunchPipeline function doesn't need to allocate every channel, and can be rewritten like this:

```
func LaunchPipeline(amount int) int {
    return <-sum(power(generator(amount)))
}
```

The result of the generator function is passed directly to the power function and the result of power to sum functions.

Final words on the Pipeline pattern

With the Pipeline pattern, we can create really complex concurrent workflows in a very easy way. In our case, we created a linear workflow, but it could also have conditionals, pools, and fan-in and fan-out behavior. We will see some of these in the following chapter.

Summary

Concurrency design patterns are a step forward in difficulty, and take some time to grasp. Our biggest mistake as concurrent programmers is thinking in terms of parallelism (How can I make this parallel? or How can I run this in a new thread?) instead of in terms of concurrent structures.

Pure functions (functions that will always produce the same output (given the same input) without affecting anything outside their scope) help in this design.

Concurrent programming requires practice and more practice. Go makes it easy once you understand the basic primitives. Diagrams can help you to understand the possible flow of data, but the best way of understanding it all is simply to practice.

In the following chapter, we will see how to use a pool of pipeline workers to do some work instead of having a unique pipeline. Also, we will learn how to create the publish/subscriber pattern in a concurrent structure and see how different the same pattern can be when we build by using concurrency.

10

Concurrency Patterns - Workers Pool and Publish/Subscriber Design Patterns

We have reached the final chapter of the book, where we will discuss a couple of patterns with concurrent structures. We will explain every step in detail so you can follow the examples carefully.

The idea is to learn about patterns to design concurrent applications in idiomatic Go. We are using channels and Goroutines heavily, instead of locks or sharing variables.

- We will look at one way to develop a pool of workers. This is useful to control the number of Goroutines in an execution.
- The second example is a rewrite of the Observer pattern, which we saw on *Chapter 7, Behavioral Patterns - Visitor, State, Mediator, and Observer Design Patterns*, written with a concurrent structure. With this example we'll dig a bit more into the concurrent structures and look at how they can differ from a common approach.

Workers pool

One problem we may face with some of the previous approaches to concurrency is their unbounded context. We cannot let an app create an unlimited amount of Goroutines. Goroutines are light, but the work they perform could be very heavy. A workers pool helps us to solve this problem.

Description

With a pool of workers, we want to bound the amount of Goroutines available so that we have a deeper control of the pool of resources. This is easy to achieve by creating a channel for each worker and having workers with either an idle or busy status. The task can seem daunting, but it's not at all.

Objectives

Creating a Worker pool is all about resource control: CPU, RAM, time, connections, and so on. The workers pool design pattern helps us to do the following:

- Control access to shared resources using quotas
- Create a limited amount of Goroutines per app
- Provide more parallelism capabilities to other concurrent structures

A pool of pipelines

In the previous chapter, we saw how to work with a pipeline. Now we will launch a bounded number of them so that the Go scheduler can try to process requests in parallel. The idea here is to control the number of Goroutines, stop them gracefully when the app has finished, and maximize parallelism using a concurrent structure without race conditions.

The pipeline we will use is similar to the one we used in the previous chapter, where we were generating numbers, raising them to the power of 2, and summing the final results. In this case, we are going to pass strings to which we will append and prefix data.

Acceptance criteria

In business terms, we want something that tells us that, worker has processed a request, a predefined ending, and incoming data parsed to uppercase:

1. When making a request with a string value (any), it must be uppercase.
2. Once the string is uppercase, a predefined text must be appended to it. This text should not be uppercase.
3. With the previous result, the worker ID must be prefixed to the final string.
4. The resulting string must be passed to a predefined handler.

We haven't talked about how to do it technically, just what the business wants. With the entire description, we'll at least have workers, requests, and handlers.

Implementation

The very beginning is a request type. According to the description, it must hold the string that will enter the pipeline as well as the handler function:

```
// workers_pipeline.go file
type Request struct {
    Data    interface{}
    Handler RequestHandler
}
```

Where is the return? We have a `Data` field of type `interface{}` so we can use it to pass a string. By using an interface, we can reuse this type for a `string`, an `int`, or a `struct` data type. The receiver is the one who must know how to deal with the incoming interface.

The `Handler` field has the type `Request handler`, which we haven't defined yet:

```
type RequestHandler func(interface{})
```

A request handler is any function that accepts an interface as its first argument, and returns nothing. Again, we see the `interface{}`, where we would usually see a `string`. This is one of the receivers we mentioned previously, which we'll need to cast the incoming result.

So, when sending a request, we must fill it with some value in the `Data` field and implement a handler; for example:

```
func NewStringRequest(s string, id int, wg *sync.WaitGroup) Request {
    return Request{
        Data: "Hello", Handler: func(i interface{})
```

```
{  
    defer wg.Done()  
    s, ok := i.(string)  
    if !ok{  
        log.Fatal("Invalid casting to string")  
    }  
    fmt.Println(s)  
}  
}  
}
```

The handler is defined by using a closure. We again check the type if the interface (and we defer the call to the `Done()` method at the end). In case of an improper interface, we simply print its contents and return. If the casting is OK, we also print them, but here is where we will usually do something with the result of the operation; we have to use type casting to retrieve the contents of the `interface{}` (which is a string). This must be done in every step in the pipeline, although it will introduce a bit of overhead.

Now we need a type that can handle `Request` types. Possible implementations are virtually infinite, so it is better to define an interface first:

```
// worker.go file  
type WorkerLauncher interface {  
    LaunchWorker(in chan Request)  
}
```

The `WorkerLauncher` interface must implement only the `LaunchWorker(chan Request)` method. Any type that implements this interface will have to receive a channel of `Request` type to satisfy it. This channel of the `Request` type is the single entrance point to the pipeline.

The dispatcher

Now, to launch workers in parallel and handle all the possible incoming channels, we'll need something like a dispatcher:

```
// dispatcher.go file  
type Dispatcher interface {  
    LaunchWorker(w WorkerLauncher)  
    MakeRequest(Request)  
    Stop()  
}
```

A Dispatcher interface can launch an injected WorkerLaunchers type in its own LaunchWorker method. The Dispatcher interface must use the LaunchWorker method of any of the WorkerLauncher types to initialize a pipeline. This way we can reuse the Dispatcher interface to launch many types of WorkerLaunchers.

When using MakeRequest (Request), the Dispatcher interface exposes a nice method to inject a new Request into the workers pool.

Finally, the user must call stop when all Goroutines must be finished. We must handle graceful shutdown in our apps, and we want to avoid Goroutine leaks.

We have enough interfaces, so let's start with the dispatcher which is a bit less complicated:

```
type dispatcher struct {
    inCh chan Request
}
```

Our dispatcher structure stores a channel of Request type in one of its fields. This is going to be the single point of entrance for requests in any pipeline. We said that it must implement three methods, as follows:

```
func (d *dispatcher) LaunchWorker(id int, w WorkerLauncher) {
    w.LaunchWorker(d.inCh)
}

func (d *dispatcher) Stop() {
    close(d.inCh)
}

func (d *dispatcher) MakeRequest(r Request) {
    d.inCh <- r
}
```

In this example, the Dispatcher interface doesn't need to do anything special to itself before launching a worker, so the LaunchWorker method on the Dispatcher simply executes the LaunchWorker method of the incoming WorkerLauncher, which also has a LaunchWorker method to initiate itself. We have previously defined that a WorkerLauncher type needs at least an ID and a channel for incoming requests, so that's what we are passing through.

It may seem unnecessary to implement the LaunchWorker method in the Dispatcher interface. In different scenarios, it could be interesting to save running worker IDs in the dispatcher to control which ones are up or down; the idea is to hide launching implementation details. In this case, the Dispatcher interface is merely acting as a Facade design pattern hiding some implementation details from the user.

The second method is `Stop`. It closes the incoming requests channel, provoking a chain reaction. We saw in the pipeline example that, when closing the incoming channel, each for-range loop within the Goroutines breaks and the Goroutine is also finished. In this case, when closing a shared channel, it will provoke the same reaction, but in every listening Goroutine, so all pipelines will be stopped. Cool, huh?

Request implementation is very simple; we just pass the request in the argument to the channel of incoming requests. The Goroutine will block there forever until the opposite end of the channel retrieves the request. Forever? That seems like a lot if something happens. We can introduce a timeout, as follows:

```
func (d *dispatcher) MakeRequest(r Request) {
    select {
        case d.inCh <- r:
        case <-time.After(time.Second * 5):
            return
    }
}
```

If you remember from previous chapters, we can use `select` to control which operation is performed over a channel. Like a `switch` case, just one operation can be executed. In this case, we have two different operations: sending and receiving.

The first case is a sending operation--try to send this, and it will block there until someone takes the value in the opposite side of the channel. Not a huge improvement, then. The second case is a receiving operation; it will be triggered after 5 seconds if the upper request can't be sent successfully, and the function will return. It would be very convenient to return an error here, but to make things simple, we will leave it empty

Finally, in the dispatcher, for convenience, we will define a `Dispatcher` creator:

```
func NewDispatcher(b int) Dispatcher {
    return &dispatcher{
        inCh:make(chan Request, b),
    }
}
```

By using this function instead of creating the dispatcher manually, we can simply avoid small mistakes, such as forgetting to initialize the channel field. As you can see, the `b` argument refers to the buffer size in the channel.

The pipeline

So, our dispatcher is done and we need to develop the pipeline described in the acceptance criteria. First, we need a type to implement the `WorkerLauncher` type:

```
// worker.go file
type PrefixSuffixWorker struct {
    id int
    prefixS string
    suffixS string
}

func (w *PrefixSuffixWorker) LaunchWorker(i int, in chan Request) {}
```

The `PrefixSuffixWorker` variable stores an ID, a string to prefix, and another string to suffix the incoming data of the `Request` type. So, the values to prefix and append will be static in these fields, and we will take them from there.

We will implement the `LaunchWorker` method later and begin with each step in the pipeline. According to *first acceptance criteria*, the incoming string must be uppercase. So, the uppercase method will be the first step in our pipeline:

```
func (w *PrefixSuffixWorker) uppercase(in <-chan Request) <-chan
Request {
    out := make(chan Request)

    go func() {
        for msg := range in {
            s, ok := msg.Data.(string)

            if !ok {
                msg.handler(nil)
                continue
            }

            msg.Data = strings.ToUpper(s)

            out <- msg
        }
    }

    close(out)
}()

return out
}
```

Good. As in the previous chapter, a step in the pipeline accepts a channel of incoming data and returns a channel of the same type. It has a very similar approach to the examples we developed in the previous chapter. This time, though, we aren't using package functions, and uppercase is part of the `PrefixSuffixWorker` type and the incoming data is a struct instead of an `int`.

The `msg` variable is a `Request` type and it will have a handler function and data in the form of an interface. The `Data` field should be a string, so we type cast it before using it. When type casting a value, we will receive the same value with the requested type and a `true` or `false` flag (represented by the `ok` variable). If the `ok` variable is `false`, the cast could not be done and we won't throw the value down the pipeline. We stop this `Request` here by sending a `nil` to the handler (which will also provoke a type-casting error).

Once we have a nice string in the `s` variable, we can uppercase it and store it again in the `Data` field to send down the pipeline to the next step. Be aware that the value will be sent as an interface again, so the next step will need to cast it again. This is the downside of using this approach.

With the first step done, let's continue with the second. According to the *second acceptance criteria* now, a predefined text must be appended. This text is the one stored in the `suffixS` field:

```
func (w *PrefixSuffixWorker) append(in <-chan Request) <-chan Request {
    out := make(chan Request)
    go func() {
        for msg := range in {
            uppercaseString, ok := msg.Data.(string)

            if !ok {
                msg.handler(nil)
                continue
            }
            msg.Data = fmt.Sprintf("%s%s", uppercaseString, w.suffixS)
            out <- msg
        }
        close(out)
    }()
    return out
}
```

The `append` function has the same structure as the `uppercase` function. It receives and returns a channel of incoming requests, and launches a new Goroutine that iterates over the incoming channel until it is closed. We need to type cast the incoming value, as mentioned previously.

In this step in the pipeline the incoming string is uppercase (after doing a type assertion). To append any text to it, we just need to use the `fmt.Sprintf()` function, as we have done many times before, which formats a new string with the provided data. In this case, we pass the value of the `prefixS` field as the second value, to append it to the end of the string.

Just the last step in the pipeline is missing, the prefix operation:

```
func (w *PrefixSuffixWorker) prefix(in <-chan Request) {
    go func() {
        for msg := range in {
            uppercasedStringWithSuffix, ok := msg.Data.(string)

            if !ok {
                msg.handler(nil)
                continue
            }

            msg.handler(fmt.Sprintf("%s%s", w.prefixS,
uppercasedStringWithSuffix))
        }
    }()
}
```

What's calling your attention in this function? Yes, it doesn't return any channel now. We could have done this entire pipeline in two ways. I suppose you have realized that we have used a `Future` handler function to execute with the final result in the pipeline. A second approach would be to pass a channel to return the data back to its origin. In some cases, a `Future` would be enough, while in others it could be more convenient to pass a channel so that it can be connected to a different pipeline (for example).

In any case, the structure of a step in a pipeline must be very familiar to you already. We cast the value, check the result of the casting, and send nil to the handler if anything went wrong. But, in case everything was OK, the last thing to do is to format the text again to place the `prefixS` field at the beginning of the text, to send the resulting string back to the origin by calling the request's handler.

Now, with our worker almost finished, we can implement the `LaunchWorker` method:

```
func (w *PrefixSuffixWorker) LaunchWorker(in chan Request) {
    w.prefix(w.append(w.uppercase(in)))
}
```

That's all for workers! We simply pass the returning channels to the next steps in the Pipeline, as we did in the previous chapter. Remember that the pipeline is executed from inside to outside of the calls. So, what's the order of execution of any incoming data to the pipeline?

1. The data enters the pipeline through the Goroutine launched in the uppercase method.
2. Then, it goes to the Goroutine launched in `append`.
3. Finally, it enters the Goroutine launched in `prefix` method, which doesn't return anything but executes the handler after prefixing the incoming string with more data.

Now we have a full pipeline and a dispatcher of pipelines. The dispatcher will launch as many instances of the pipelines as we want to route the incoming requests to any available worker.

If none of the workers takes the request within 5 seconds, the request is lost.

Let's use this library in a small app.

An app using the workers pool

We will launch three workers of our defined pipeline. We use the `NewDispatcher` function to create the dispatcher and the channel that will receive all requests. This channel has a fixed buffer, which will be able to store up to 100 incoming messages before blocking:

```
// workers_pipeline.go
func main() {
    bufferSize := 100
    var dispatcher Dispatcher = NewDispatcher(bufferSize)
```

Then, we will launch the workers by calling the `LaunchWorker` method in the `Dispatcher` interface three times with an already filled `WorkerLauncher` type:

```
workers := 3
for i := 0; i < workers; i++ {
    var w WorkerLauncher = &PrefixSuffixWorker{
        prefixS: fmt.Sprintf("WorkerID: %d -> ", i),
        suffixS: " World",
        id:i,
    }
    dispatcher.LaunchWorker(w)
}
```

Each `WorkerLauncher` type is an instance of `PrefixSuffixWorker`. The prefix will be a small text showing the worker ID and the suffix text world.

At this point, we have three workers with three Goroutines, each running concurrently and waiting for messages to arrive:

```
requests := 10

var wg sync.WaitGroup
wg.Add(requests)
```

We will make 10 requests. We also need a WaitGroup to properly synchronize the app so that it doesn't exit too early. You can find yourself using WaitGroups quite a lot when dealing with concurrent applications. For 10 requests, we'll need to wait for 10 calls to the `Done()` method, so we call the `Add()` method with a *delta* of 10. It's called delta because you can also pass a -5 later to leave it in five requests. In some situations, it can be useful:

```
for i := 0; i < requests; i++ {
    req := NewStringRequest("(Msg_id: %d) -> Hello", i, &wg)
    dispatcher.MakeRequest(req)
}

dispatcher.Stop()

wg.Wait()
}
```

To make requests, we will iterate a `for` loop. First, we create a `Request` using the function `NewStringRequest` that we wrote at the beginning of the Implementation section. In this value, the `Data` field will be the text we'll pass down the pipeline, and it will be the text that is "in the middle" of the appending and suffixing operation. In this case, we will send the message number and the word hello.

Once we have a request, we call the `MakeRequest` method with it. After all requests have been done, we stop the dispatcher that, as explained previously, will provoke a chain reaction that will stop all Goroutines in the pipeline.

Finally, we wait for the group so that all calls to the `Done()` method are received, which signals that all operations have been finished. It's time to try it out:

```
go run *
WorkerID: 1 -> (MSG_ID: 0) -> HELLO World
WorkerID: 0 -> (MSG_ID: 3) -> HELLO World
WorkerID: 0 -> (MSG_ID: 4) -> HELLO World
WorkerID: 0 -> (MSG_ID: 5) -> HELLO World
WorkerID: 2 -> (MSG_ID: 2) -> HELLO World
```

```
WorkerID: 1 -> (MSG_ID: 1) -> HELLO World
WorkerID: 0 -> (MSG_ID: 6) -> HELLO World
WorkerID: 2 -> (MSG_ID: 9) -> HELLO World
WorkerID: 0 -> (MSG_ID: 7) -> HELLO World
WorkerID: 0 -> (MSG_ID: 8) -> HELLO World
```

Let's analyze the first message:

1. This would be zero, so the message sent is (Msg_id: 0) -> Hello.
2. Then, the text is uppercased, so now we have (MSG_ID: 0) -> HELLO.
3. After uppercasing an append operation with the text world (note the space at the beginning of the text) is done. This will give us the text (MSG_ID: 0) -> HELLO World.
4. Finally, the text WorkerID: 1 (in this case, the first worker took the task, but it could be any of them) is appended to the text from step 3 to give us the full returned message, WorkerID: 1 -> (MSG_ID: 0) -> HELLO World.

No tests?

Concurrent applications are difficult to test, especially if you are doing networking operations. It can be difficult, and code can change a lot just to test it. In any case, it is not justifiable to not perform tests. In this case, it is not especially difficult to test our small app. Create a test and copy/paste the contents of the main function there:

```
//workers_pipeline.go file
package main

import "testing"

func Test_Dispatcher(t *testing.T){
    //pasted code from main function
    bufferSize := 100
    var dispatcher Dispatcher = NewDispatcher(bufferSize)
    workers := 3
    for i := 0; i < workers; i++ {
        var w WorkerLauncher = &PrefixSuffixWorker{
            prefixS: fmt.Sprintf("WorkerID: %d -> ", i), suffixS: " World",
        id: i,
        dispatcher.LaunchWorker(w)
    }
    //Simulate Requests
    requests := 10
```

```
var wg
sync.WaitGroup
wg.Add(requests)
}
```

Now we have to rewrite our handler to test that the returned contents are the ones we are expecting. Go to the `for` loop to modify the function that we are passing as a handler on each Request:

```
for i := 0; i < requests; i++ {
    req := Request{
        Data: fmt.Sprintf("(Msg_id: %d) -> Hello", i),
        handler: func(i interface{}) {
            {
                s, ok := i.(string)
                defer wg.Done()
                if !ok {
                    t.Fail()
                }
                ok, err := regexp.Match(`WorkerID\: \d* -> \MSG_ID: \d*` ->
[A-Z]*\sWorld`, []byte(s))
                if !ok || err != nil {
                    t.Fail()
                }
            },
        }
    }
    dispatcher.MakeRequest(req)
}
```

We are going to use regular expressions to test the business. If you are not familiar with regular expressions, they are a quite powerful feature that help you to match content within a string. If you remember in our exercises when we were using the `strings` package. `Contains` is the function to find a text inside a string. We can also do it with regular expressions.

The problem is that regular expressions are quite expensive and consume a lot of resources.

We are using the `Match` function of the `regexp` package to provide a template to match. Our template is `WorkerID\:\: \d* -> \ (MSG_ID\: \d\:) -> [A-Z]*\sWorld` (without quotes). Specifically, it describes the following:

- A string that has the content `WorkerID\: \d* -> (MSG_ID\: \d*)`, here "`\d*`" indicates any digit written zero or more times, so it will match `WorkerID: 10 -> (MSG_ID: 1)` and "`WorkerID: 1 -> (MSG_ID: 10)`".
- "`\) -> [A-Z]*\sWorld`" (parentheses must be escaped using backslashes). "`*`" means any uppercase character written zero or more times, so "`\s`" is a white space and it must finish with the text `World`, so `) -> HELLO World`" will match, but `) -> Hello World`" won't, because "Hello" must be all uppercase.

Running this test gives us the following output:

```
go test -v .
==== RUN Test_Dispatcher
--- PASS: Test_Dispatcher (0.00s)
PASS
ok
```

Not bad, but we aren't testing that code is being executed concurrently, so this is more a business test than a unit test. Concurrency testing would force us to write the code in a completely different manner to check that it is creating the proper amount of Goroutines and the pipeline is following the expected workflow. This is not bad, but it's quite complex, and outside of the context of this book.

Wrapping up the Worker pool

With the workers pool, we have our first complex concurrent application that can be used in real-world production systems. It also has room to improve, but it is a very good design pattern to build concurrent bounded apps.

It is key that we always have the number of Goroutines that are being launched under control. While it's easy to launch thousands to achieve more parallelism in an app, we must be very careful that they don't have code that can hang them in an infinite loop, too.

With the workers pool, we can now fragment a simple operation in many parallel tasks. Think about it; this could achieve the same result with one simple call to `fmt.Printf`, but we have done a pipeline with it; then, we launched few instances of this pipeline and finally, distributed the workload between all those pipes.

Concurrent Publish/Subscriber design pattern

In this section, we will implement the Observer design pattern that we showed previously on Behavioral patterns, but with a concurrent structure and thread safety.

Description

If you remember from the previous explanation, the Observer pattern maintains a list of observers or subscribers that want to be notified of a particular event. In this case, each subscriber is going to run in a different Goroutine as well as the publisher. We will have new problems with building this structure:

- Now, the access to the list of subscribers must be serialized. If we are reading the list with one Goroutine, we cannot be removing a subscriber from it or we will have a race.
- When a subscriber is removed, the subscriber's Goroutine must be closed too, or it will keep iterating forever and we will run into Goroutine leaks.
- When stopping the publisher, all subscribers must stop their Goroutines, too.

Objectives

The objectives of this publish/subscriber are the same as the ones we wrote on the Observer pattern. The difference here is the way we will develop it. The idea is to make a concurrent structure to achieve the same functionality, which is as follows:

- Providing an event-driven architecture where one event can trigger one or more actions
- Uncoupling the actions that are performed from the event that triggers them
- Providing more than one source event that triggers the same action

The idea is to uncouple senders from receivers, hiding from the sender the identity of the receivers that will process its event, and hiding the receivers from the number of senders that can communicate with them.

In particular, if I develop a click in a button in some application, it could do something (such as log us in somewhere). Weeks later, we might decide to make it show a popup, too. If, every time we want to add some functionality to this button, we have to change the code where it handles the click action, that function will become huge and not very portable to other projects. If we use a publisher and one observer for every action, the click function only needs to publish one single event using a publisher, and we will just write subscribers to this event every time we want to improve the functionality. This is especially important in applications with user interfaces where many things to do in a single UI action can slow the responsiveness of an interface, completely destroying the user experience.

By using a concurrent structure to develop the Observer pattern, a UI cannot feel all the tasks that are being executed in the background if a concurrent structure is defined and the device allows us to execute parallel tasks.

Example - a concurrent notifier

We will develop a *notifier* similar to the one we developed in Chapter 7, *Behavioral Patterns - Visitor, State, Mediator, and Observer Design Patterns*. This is to focus on the concurrent nature of the structure instead of detailing too many things that have already been explained. We have developed an observer already, so we are familiar with the concept.

This particular notifier will work by passing around `interface{}` values, like in the workers pool example. This way, we can use it for more than a single type by introducing some overhead when casting on the receiver.

We will work with two interfaces now. First, a `Subscriber` interface:

```
type Subscriber interface {
    Notify(interface{}) error
    Close()
}
```

Like in the previous example, it must have a `Notify` method in the `Subscriber` interface of new events. This is the `Notify` method that accepts an `interface{}` value and returns an error. The `Close()` method, however, is new, and it must trigger whatever actions are needed to stop the Goroutine where the subscriber is listening for new events.

The second and final interface is the Publisher interface:

```
type Publisher interface {
    start()
    AddSubscriberCh() chan<- Subscriber
    RemoveSubscriberCh() chan<- Subscriber
    PublishingCh() chan<- interface{}
    Stop()
}
```

The Publisher interface has the same actions we already know for a publisher but to work with channels. The AddSubscriberCh and RemoveSubscriberCh methods accepts a Subscriber interface (any type that satisfies the Subscriber interface). It must have a method to publish messages and a Stop method to stop them all (publisher and subscriber Goroutines)

Acceptance criteria

Requirements between this example and the one in the *Chapter 7, Behavioral patterns - Visitor, State, Mediator, and Observer Design Patterns* must not change. The objective in both examples is the same so the requirements must also be the same. In this case, our requirements are technical, so we actually need to add some more acceptance criteria:

1. We must have a publisher with a PublishingCh method that returns a channel to send messages through and triggers a Notify method on every observer subscribed.
2. We must have a method to add new subscribers to the publisher.
3. We must have a method to remove new subscribers from the publisher.
4. We must have a method to stop a subscriber.
5. We must have a method to stop a Publisher interface that will also stop all subscribers.
6. All inter Goroutine communication must be synchronized so that no Goroutine is locked waiting for a response. In such cases, an error is returned after the specified timeout period has passed.

Well, these criteria seem quite daunting. We have left out some requirements that would add even more complexity, such as removing non-responding subscribers or checks to monitor that the publisher Goroutine is always on.

Unit test

We have mentioned previously that testing concurrent applications can be difficult. With the correct mechanism, it still can be done, so let's see how much we can test without big headaches.

Testing subscriber

Starting with subscribers, which seem to have a more encapsulated functionality, the first subscriber must print incoming messages from the publisher to an `io.Writer` interface. We have mentioned that the subscriber has an interface with two methods,

`Notify(interface{}) error` and the `Close()` method:

```
// writer_sub.go file
package main

import "errors"

type writerSubscriber struct {
    id int
    Writer io.Writer
}

func (s *writerSubscriber) Notify(msg interface{}) error {
    return errors.New("Not implemented yet")
}
func (s *writerSubscriber) Close() {}
```

OK. This is going to be our `writer_sub.go` file. Create the corresponding test file, called the `writer_sub_test.go` file:

```
package main
func TestStdoutPrinter(t *testing.T) {
```

Now, the first problem we have is that the functionality prints to the `stdout`, so there's no return value to check. We can solve it in three ways:

- Capturing the `stdout` method.
- Injecting an `io.Writer` interface to print to it. This is the preferred solution, as it makes the code more manageable.
- Redirecting the `stdout` method to a different file.

We'll take the second approach. Redirection is also a possibility. The `os.Stdout` is a pointer to an `os.File` type, so it involves replacing this file with one we control, and reading from it:

```
func TestWriter(t *testing.T) {
    sub := NewWriterSubscriber(0, nil)
```

The `NewWriterSubscriber` subscriber isn't defined yet. It must help in the creation of this particular subscriber, returning a type that satisfies the `Subscriber` interface, so let's quickly declare it on the `writer_sub.go` file:

```
func NewWriterSubscriber(id int, out io.Writer) Subscriber {
    return &writerSubscriber{}
}
```

Ideally, it must accept an ID and an `io.Writer` interface as the destination for its writes. In this case, we need a custom `io.Writer` interface for our test, so we'll create a `mockWriter` on the `writer_sub_test.go` file for it:

```
type mockWriter struct {
    testingFunc func(string)
}

func (m *mockWriter) Write(p []byte) (n int, err error) {
    m.testingFunc(string(p))
    return len(p), nil
}
```

The `mockWriter` structure will accept a `testingFunc` as one of its fields. This `testingFunc` field accepts a string that represents the bytes written to the `mockWriter` structure. To implement an `io.Writer` interface, we need to define a `Write([]byte) (int, error)` method. In our definition, we pass the contents of `p` as a string (remember that we always need to return the bytes read and an error, or not, on every `Write` method). This approach delegates the definition of `testingFunc` to the scope of the test.

We are going to call the `Notify` method on the `Subscriber` interface, which must write on the `io.Writer` interface like the `mockWriter` structure. So, we'll define the `testingFunc` of a `mockWriter` structure before calling the `Notify` method:

```
// writer_sub_test.go file
func TestPublisher(t *testing.T) {
    msg := "Hello"

    var wg sync.WaitGroup
    wg.Add(1)
```

```
    stdoutPrinter := sub.(*writerSubscriber)
    stdoutPrinter.Writer = &mockWriter{
        testingFunc: func(res string) {
            if !strings.Contains(res, msg) {
                t.Fatal(fmt.Errorf("Incorrect string: %s", res))
            }
            wg.Done()
        },
    }
}
```

We will send the `Hello` message. This also means that whatever the `Subscriber` interface does, it must eventually print the `Hello` message on the provided `io.Writer` interface.

So if, eventually, we receive a string on the testing function, we'll need to synchronize with the `Subscriber` interface to avoid race conditions on tests. That's why we use so much `WaitGroup`. It's a very handy and easy-to-use type to handle this scenario. One `Notify` method call will need to wait for one call to the `Done()` method, so we call the `Add(1)` method (with one unit).

Ideally, the `NewWriterSubscriber` function must return an interface, so we need to type assert it to the type we are working with during the test, in this case, the `stdoutPrinter` method. I have omitted error checking when doing the casting on purpose, just to make things easier. Once we have a `writerSubscriber` type, we can access its `Write` field to replace it with the `mockWriter` structure. We could have directly passed an `io.Writer` interface on the `NewWriterSubscriber` function, but we wouldn't cover the scenario where a nil object is passed and it sets the `os.Stdout` instance to a default value.

So, the testing function will eventually receive a string containing what was written by the subscriber. We just need to check if the received string, the one that the `Subscriber` interface will receive, prints the word `Hello` at some point and nothing better than `strings.Contains` function for it. Everything is defined under the scope of the testing function, so we can use the value of the `t` object to also signal that the test has failed.

Once we have done the checking, we must call to the `Done()` method to signal that we have already tested the expected result:

```
err := sub.Notify(msg)
if err != nil {
    t.Fatal(err)
}

wg.Wait()
sub.Close()
}
```

We must actually call the `Notify` and `Wait` methods for the call to the `Done` method to check that everything was correct.



Did you realize that we have defined the behavior on tests more or less in reverse? This is very common in concurrent apps. It can be confusing sometimes, as it becomes difficult to know what a function could be doing if we can't follow calls linearly, but you get used to it quite quickly. Instead of thinking "it does this, then this, then that," it's more like "this will be called when executing that." This is also because the order of execution in a concurrent application is unknown until some point, unless we use synchronization primitives (such as `WaitGroups` and channels) to pause execution at certain moments.

Let's execute the test for this type now:

```
go test -cover -v -run=TestWriter .
==== RUN    TestWriter
--- FAIL: TestWriter (0.00s)
    writer_sub_test.go:40: Not implemented yet
FAIL
coverage: 6.7% of statements
exit status 1
FAIL
```

It has exited fast but it has failed. Actually, the call to the `Done()` method has not been executed, so it would be nice to change the last part of our test to this instead:

```
err := sub.Notify(msg)
if err != nil {
    wg.Done()
    t.Error(err)
}
wg.Wait()
sub.Close()
```

Now, it doesn't stop execution because we are calling the `Error` function instead of the `Fatal` function, but we call the `Done()` method and the test ends where we prefer it to end, after the `Wait()` method is called. You can try to run the tests again, but the output will be the same.

Testing publisher

We have already seen a `Publisher` interface and the type that will satisfy which was the `publisher` type. The only thing we know for sure is that it will need some way to store subscribers, so it will at least have a `Subscribers` slice:

```
// publisher.go type
type publisher struct {
    subscribers []Subscriber
}
```

To test the `publisher` type, we will also need a mock for the `Subscriber` interface:

```
// publisher_test.go
type mockSubscriber struct {
    notifyTestingFunc func(msg interface{})
    closeTestingFunc func()
}

func (m *mockSubscriber) Close() {
    m.closeTestingFunc()
}

func (m *mockSubscriber) Notify(msg interface{}) error {
    m.notifyTestingFunc(msg)
    return nil
}
```

The `mockSubscriber` type must implement the `Subscriber` interface, so it must have a `Close()` and a `Notify(interface{}) error` method. We can embed an existing type that implements it, such as, the `writerSubscriber`, and override just the method that is interesting for us, but we will need to define both, so we won't embed anything.

So, we need to override the `Notify` and `Close` methods in this case to call the testing functions stored on the fields of the `mockSubscriber` type:

```
func TestPublisher(t *testing.T) {
    msg := "Hello"

    p := NewPublisher()
```

First of all, we will be sending messages through channels directly, this could lead to potential unwanted deadlocks so the first thing to define is a panic handler for cases such as, sending to close channels or no Goroutines listening on a channel. The message we will send to subscribers is `Hello`. So, each subscriber that has been received using the channel returned by the `AddSubscriberCh` method must receive this message. We will also use a `New` function to create Publishers, called `NewPublisher`. Change the `publisher.go` file now to write it:

```
// publisher.go file
func NewPublisher() Publisher {
    return &publisher{}
}
```

Now we'll define the `mockSubscriber` to add it to the publisher list of known subscribers. Back to the `publisher_test.go` file:

```
var wg sync.WaitGroup

sub := &mockSubscriber{
    notifyTestingFunc: func(msg interface{}) {
        defer wg.Done()

        s, ok := msg.(string)
        if !ok {
            t.Fatal(errors.New("Could not assert result"))
        }

        if s != msg {
            t.Fail()
        }
    },
    closeTestingFunc: func() {
        wg.Done()
    },
}
```

As usual, we start with a `WaitGroup`. First, testing the function in our subscriber defers a call to the `Done()` method at the end of its execution. Then it needs to type cast `msg` variable because it's coming as an interface. Remember that this way, we can use the `Publisher` interface with many types by introducing the overhead of the type assertion. This is done on line `s, ok := msg.(string)`.

Once we have type cast `msg` to a string, `s`, we just need to check if the value received in the subscriber is the same as the value we sent, or fail the test if not:

```
p.AddSubscriberCh() <- sub
wg.Add(1)

p.PublishingCh() <- msg
wg.Wait()
```

We add the `mockSubscriber` type using the `AddSubscriberCh` method. We publish our message just after getting ready, by adding one to the `WaitGroup`, and just before setting the `WaitGroup` to wait so that the test doesn't continue until the `mockSubscriber` type calls the `Done()` method.

Also, we need to check if the number of the `Subscriber` interface has grown after calling the `AddSubscriberCh` method, so we'll need to get the concrete instance of publisher on the test:

```
pubCon := p.(*publisher)
if len(pubCon.subscribers) != 1 {
    t.Error("Unexpected number of subscribers")
}
```

Type assertion is our friend today! Once we have the concrete type, we can access the underlying slice of subscribers for the `Publisher` interface. The number of subscribers must be 1 after calling the `AddSubscriberCh` method once, or the test will fail. The next step is to check just the opposite--when we remove a `Subscriber` interface, it must be taken from this list:

```
wg.Add(1)
p.RemoveSubscriberCh() <- sub
wg.Wait()

//Number of subscribers is restored to zero
if len(pubCon.subscribers) != 0 {
    t.Error("Expected no subscribers")
}

p.Stop()
}
```

The final step in our test is to stop the publisher so no more messages can be sent and all the Goroutines are stopped.

The test is finished, but we can't run tests until the publisher type has all the methods implemented; this must be the final result:

```
type publisher struct {
    subscribers []Subscriber
    addSubCh    chan Subscriber
    removeSubCh chan Subscriber
    in          chan interface{}
    stop        chan struct{}
}

func (p *publisher) AddSubscriberCh() chan<- Subscriber {
    return nil
}

func (p *publisher) RemoveSubscriberCh() chan<- Subscriber {
    return nil
}

func (p *publisher) PublishingCh() chan<- interface{} {
    return nil
}

func (p *publisher) Stop(){}

```

With this empty implementation, nothing good can happen when running the tests:

```
go test -cover -v -run=TestPublisher .
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan receive]:
testing.(*T).Run(0xc0420780c0, 0x5244c6, 0xd, 0x5335a0, 0xc042037d20)
    /usr/local/go/src/testing/testing.go:647 +0x31d
testing.RunTests.func1(0xc0420780c0)
    /usr/local/go/src/testing/testing.go:793 +0x74
testing.tRunner(0xc0420780c0, 0xc042037e10)
    /usr/local/go/src/testing/testing.go:610 +0x88
testing.RunTests(0x5335b8, 0x5ada40, 0x2, 0x2, 0x40d7e9)
    /usr/local/go/src/testing/testing.go:799 +0x2fc
testing.(*M).Run(0xc042037ed8, 0xc04200a4f0)
    /usr/local/go/src/testing/testing.go:743 +0x8c
main.main()
    go-design-patterns/concurrency_3/pubsub/_test/_testmain.go:56 +0xcd

goroutine 5 [chan send (nil chan)]:
go-design-patterns/concurrency_3/pubsub.TestPublisher(0xc042078180)
    go-design-patterns/concurrency_3/pubsub/publisher_test.go:55 +0x372
testing.tRunner(0xc042078180, 0x5335a0)
```

```
/usr/local/go/src/testing/testing.go:610 +0x88
created by testing.(*T).Run
/usr/local/go/src/testing/testing.go:646 +0x2f3
exit status 2
FAIL go-design-patterns/concurrency_3/pubsub    1.587s
```

Yes it has failed but, it's not a controlled fail at all. This was done on purpose to show a couple of things to be careful of in Go. First of all, the error produced in this test is a **fatal** error, which usually points to a bug in the code. This is important because while a **panic** error can be recovered, you cannot do the same with a fatal error.

In this case, the error is telling us the problem: `goroutine 5 [chan send (nil chan)]`, a nil channel so it's actually a bug in our code. How can we solve this? Well, this is also interesting.

The fact that we have a `nil` channel is caused by the code we wrote to compile unit tests but this particular error won't be raised once the appropriate code is written (because we'll never return a nil channel in this case). We could return a channel that is never used we cause a fatal error with a deadlock, which wouldn't be any progress at all either.

An idiomatic way to solve it would be to return a channel and an error so that you can have an error package with a type implementing the `Error` interface that returns a specific error such as `NoGoroutinesListening` or `ChannelNotCreated`. We have already seen many of this implementations so we'll leave these as an exercise to the reader and we will move forward to maintain focus on the concurrent nature of the chapter.

Nothing surprising there, so we can move to the implementation phase.

Implementation

To recall, the `writerSubscriber` must receive messages that it will write on a type that satisfies the `io.Writer` interface.

So, where do we start? Well, each subscriber will run its own Goroutine, and we have seen that the best method to communicate with a Goroutine is a channel. So, we will need a field with a channel in the `Subscriber` type. We can use the same approach as in pipelines to end with the `NewWriterSubscriber` function and the `writerSubscriber` type:

```
type writerSubscriber struct {
    in     chan interface{}
    id    int
    Writer io.Writer
}
```

```
func NewWriterSubscriber(id int, out io.Writer) Subscriber {
    if out == nil {
        out = os.Stdout
    }

    s := &writerSubscriber{
        id:      id,
        in:      make(chan interface{}),
        Writer:  out,
    }

    go func() {
        for msg := range s.in {
            fmt.Fprintf(s.Writer, "(W%d): %v\n", s.id, msg)
        }
    }()
}

return s
}
```

In the first step, if no writer is specified (the `out` argument is `nil`), the default `io.Writer` interface is `stdout`. Then, we create a new pointer to the `writerSubscriber` type with the ID passed in the first argument, the value of `out` (`os.Stdout`, or whatever came in the argument if it wasn't `nil`), and a channel called `in` to maintain the same naming as in previous examples.

Then we launch a new Goroutine; this is the launching mechanism we mentioned. Like in the pipelines, the subscriber will iterate over the `in` channel every time a new message is received and it will format its contents to a string, which also contains the ID of the current subscriber.

As we learned previously, if the `in` channel is closed, the `for range` loop will stop and that particular Goroutine will finish, so the only thing we need to do in the `Close` method is to actually close the `in` channel:

```
func (s *writerSubscriber) Close() {
    close(s.in)
}
```

OK, only the `Notify` method is left; the `Notify` method is a convenient method to manage a particular behavior when communicating, and we will use a pattern that is common in many calls:

```
func (s *writerSubscriber) Notify(msg interface{}) (err error) {
    defer func() {
        if rec := recover(); rec != nil {
```

```
        err = fmt.Errorf("%#v", rec)
    }
}()

select {
case s.in <- msg:
case <-time.After(time.Second):
    err = fmt.Errorf("Timeout\n")
}

return
}
```

When communicating with a channel, there are two behavior that we must usually control: one is waiting time and the other is when the channel is closed. The deferred function actually works for any panicking error that can occur within the function. If the Goroutine panics, it will still execute the deferred function with the `recover()` method. The `recover()` method returns an interface of whatever the error was, so in our case, we set the returning variable `error` to the formatted value returned by `recover` (which is an interface). The `"%#v"` parameter gives us most of the information about any type when formatted to a string. The returned error will be ugly, but it will contain most of the information we can extract about the error. For a closed channel, for example, it will return "send on a closed channel". Well, this seems clear enough.

The second rule is about waiting time. When we send a value over a channel, we will be blocked until another Goroutine takes the value from it (it will happen the same with a filled buffered channel). We don't want to get blocked forever, so we set a timeout period of one second by using a select handler. In short, with select we are saying: either you take the value in less than 1 second or I will discard it and return an error.

We have the `Close`, `Notify`, and `NewWriterSubscriber` methods, so we can try our test again:

```
go test -run=TestWriter -v .
==== RUN TestWriter
--- PASS: TestWriter (0.00s)
PASS
ok
```

Much better now. The `Writer` has taken the mock writer we wrote on the test and has written to it the value we pass to the `Notify` method. At the same time, `close` has probably closed the channel effectively, because the `Notify` method is returning an error after calling the `Close` method. One thing to mention is that we can't check if a channel is closed or not without interacting with it; that's why we had to defer the execution of a closure that will check the contents of the `recover()` function in the `Notify` method.

Implementing the publisher

OK, the publisher will need also a launching mechanism, but the main problems to deal with are race conditions accessing the subscriber list. We can solve this issue with a `Mutex` object from the `sync` package but we have already seen how to use this so we will use channels instead.

When using channels, we will need a channel for each action that can be considered dangerous--add a subscriber, remove a subscriber, retrieve the list of subscribers to `Notify` method them of a message, and a channel to stop all the subscribers. We also need a channel for incoming messages:

```
type publisher struct {
    subscribers []Subscriber
    addSubCh    chan Subscriber
    removeSubCh chan Subscriber
    in          chan interface{}
    stop        chan struct{}
}
```

Names are self-descriptive but, in short, subscribers maintain the list of subscribers; this is the slice that needs multiplexed access. The `addSubCh` instance is the channel to communicate with when you want to add a new subscriber; that's why it's a channel of subscribers. The same explanation applies to the `removeSubCh` channel, but this channel is to remove the subscriber. The `in` channel will handle incoming messages that must be broadcast to all subscribers. Finally, the `stop` channel must be called when we want to kill all Goroutines.

OK, let's start with the `AddSubscriberCh`, `RemoveSubscriber` and `PublishingCh` methods, which must return the channel to add and remove subscribers and the channel to send messages to all of them:

```
func (p *publisher) AddSubscriber() {
    return p.addSubCh
}
```

```
func (p *publisher) RemoveSubscriberCh() {
    return p.removeSubCh
}

func (p *publisher) PublishMessage() {
    return p.in
}
```

The `Stop()` function stops the stop channel by closing it. This will effectively spread the signal to every listening Goroutine:

```
func (p *publisher) Stop() {
    close(p.stop)
}
```

The `Stop` method, the function to stop the publisher and the subscribers, also pushes to its respective channel, called `stop`.

You may be wondering why we don't simply leave the channels available so that users push directly to this channel instead of using the proxying function. Well, the idea is that the user that integrates the library in their app doesn't have to deal with the complexity of the concurrent structure associated with the library, so they can focus on their business while maximizing performance as much as possible.

Handling channels without race conditions

Until now, we have forwarded data to the channels on the publisher but we haven't actually handled any of that data. The launcher mechanism that is going to launch a different Goroutine will handle them all.

We will create a launch method that we will execute by using the `go` keyword instead of embedding the whole function inside the `NewPublisher` function:

```
func (p *publisher) start() {
    for {
        select {
        case msg := <-p.in:
            for _, ch := range p.subscribers {
                sub.Notify(msg)
            }
        }
    }
}
```

`Launch` is a private method and we haven't tested it. Remember that private methods are usually called from public methods (the ones we have tested). Generally, if a private method is not called from a public method, it can't be called at all!

The first thing we notice with this method is that it is an infinite for loop that will repeat a select operation between many channels but only one of them can be executed each time. The first of these operations is the one that receives a new message to publish to subscribers. The `case msg := <- p.in:` code handles this incoming operation.

In this case, we are iterating over all subscribers and executing their `Notify` method. You may be wondering why we don't add the `go` keyword in front so that the `Notify` method is executed as a different Goroutine and therefore iterates much faster. Well, this because we aren't demultiplexing the actions of receiving a message and of closing the message. So, if we launch the subscriber in a new Goroutine and it is closed while the message is processed in the `Notify` method, we'll have a race condition where a message will try to be sent within the `Notify` method to a closed channel. In fact, we are considering this scenario when we develop the `Notify` method but, still, we won't control the number of Goroutines launched if we call the `Notify` method in a new Goroutine each time. For simplicity, we just call the `Notify` method, but it is a nice exercise to control the number of Goroutines waiting for a return in a `Notify` method execution. By buffering the `in` channel in each subscriber, we can also achieve a good solution:

```
case sub := <-p.addSubCh:  
    p.subscribers = append(p.subscribers, sub)
```

The next operation is what to do when a value arrives to the channel to add subscribers. In this case it's simple: we update it, appending the new value to it. While this case is executed, not other calls can be executed in this selection:

```
case sub := <-p.removeSubCh:  
    for i, candidate := range p.subscribers {  
        if candidate == sub {  
            p.subscribers = append(p.subscribers[:i],  
                p.subscribers[i+1:]...)  
            candidate.Close()  
            break  
        }  
    }
```

When a value arrives at the remove channel, the operation is a bit more complex because we have to search for the subscriber in the slice. We use a $O(N)$ approach for it, iterating from the beginning until we find it, but the search algorithm could be greatly improved. Once we find the corresponding `Subscriber` interface, we remove it from the subscribers slice and stop it. One thing to mention is that on tests, we are accessing the length of the subscribers slice directly without demultiplexing the operation. This is clearly a race condition, but generally, it isn't reflected when running the race detector.

The solution will be to develop a method just to multiplex calls to get the length of the slice, but it won't belong to the public interface. Again, for simplicity, we'll leave it like this, or this example may become too complex to handle:

```
case <-p.stop:
    for _, sub := range p.subscribers {
        sub.Close()
    }

    close(p.addSubCh)
    close(p.in)
    close(p.removeSubCh)

    return
}
}
```

The last operation to demultiplex is the `stop` operation, which must stop all Goroutines in the publisher and subscribers. Then we have to iterate through every Subscriber stored in the `subscribers` field to execute their `Close()` method, so their Goroutines are closed, too. Finally, if we return this Goroutine, it will finish, too.

OK, time to execute all tests and see how is it going:

```
go test -race .
ok
```

Not so bad. All tests have passed successfully and we have our Observer pattern ready. While the example can still be improved, it is a great example of how we must handle an Observer pattern using channels in Go. As an exercise, we encourage you to try the same example using mutexes instead of channels to control access. It's a bit easier, and will also give you an insight of how to work with mutexes.

A few words on the concurrent Observer pattern

This example has demonstrated how to take advantage of multi-core CPUs to build a concurrent message publisher by implementing the Observer pattern. While the example was long, we have tried to show a common pattern when developing concurrent apps in Go.

Summary

We have seen few approaches to develop concurrent structures that can be run in parallel. We have tried to show a few ways to solve the same problem, one without concurrency primitives and one with them. We have seen how different the publish/subscriber example written with a concurrent structure can be compared to the classic one.

We have also seen how to build a concurrent operation using a pipeline and we have parallelize it by using a worker pool, a very common Go pattern to maximize parallelism.

Both examples were simple enough to grasp, while digging as much as possible in to the nature of the Go language instead of in the problem itself.

Index

A

Abstract Factory design pattern
about 75, 83
description 75
objectives 75
vehicle factory, example 76
actor-based concurrency
versus Communicating Sequential Processes (CSP) 278
actors 216
Adapter design pattern
about 99
acceptance criteria 100
description 99
examples 104, 106, 107
implementing 103
incompatible interface, using 100
objective 100
Printer Adapter, unit testing 100, 102
requisites 100
source code 108
anonymous function 21
Apache Mesos 277
app
with workers pool 351, 353
arithmetic library 39
arrays
about 24
zero-initialization 25

B

Barrier concurrency pattern
about 311
acceptance criteria 312
advantages 320
description 311

HTTP GET aggregator 311
implementation 316, 320
integration test, writing 313, 316
objectives 311
BasicAuthMiddleware middleware 141
binary Go folder 11
buffered channels 295, 297
Builder design pattern
about 56
description 57
objectives 57
vehicle manufacturing, example 57
wrapping up 65

callback hell 287
callbacks 285, 287
chain of responsibility design pattern
about 180
closure 190, 191, 192
description 181
implementation 187, 188, 189, 192
multi-logger chain 181
objectives 181
unit testing 182, 183, 187
channels
about 293
buffered channels 295, 297
creating 293, 294, 295
directional channels 297, 298
ranging over 302, 303
select statement 298, 300
closures 22
Command design pattern
about 192
acceptance criteria 194
chain of responsibility design pattern, using 199,

- 201
description 192
example 194
examples 197, 199
implementation 195, 197
objectives 193
wrapping up 201
common channel 312
Communicating Sequential Processes (CSP)
about 276
versus actor-based concurrency 278
Composite design pattern
about 91
acceptance criteria 93
Binary Trees compositions 97
compositions, creating 93, 96
description 92
features 99
objective 92
requisites 93
used, for solving swimmer and fish problem 93
versus inheritance 98
concurrency
history 275
versus parallelism 276
concurrent multi-operation
building 333
LaunchPipeline function, implementing 339
lists, generating 338
numbers, raising to power of 2 338
reduce operation 339
concurrent publish/subscriber design pattern
about 356
acceptance criteria 358
channels, handling 371, 372, 373
conclusion 373
concurrent notifier example 357
description 356
implementation 367, 368
objectives 356
publisher, implementing 370, 371
publisher, testing 363, 364, 366, 367
subscriber, testing 359, 360, 362
unit tests, writing 359
concurrent singleton
implementation 304, 305, 308
testing 303, 304
writing 303
Concurrent Versions System (CVS) 27
ConsoleStrategy type 167
constants 16
Creational design patterns 56
customized shirts shop example
about 84
acceptance criteria 85
implementation 88
unit test 85
- ## D
- Decorator design pattern
about 131
acceptance criteria 132
description 131
example 132, 139, 140, 142, 143
Go's structural typing 145
implementation 137, 139
objectives 132
unit testing 133, 134, 135
versus Proxy design pattern 146
Delve 14
direct composition 93
directional channels 297, 298
- ## E
- embedding composition 93
encoding package 43, 45
End User License Agreement (EULA) 11
Erlang 276
errors
creating 22
handling 22
returning 22
ETCD project
about 41
URL 41
- ## F
- Facade design pattern
about 146
acceptance criteria 147

- description 146
example 147
implementing 152, 154
library, creating 154
objectives 147
unit testing 148, 149, 151
- Factory method pattern**
about 66, 75
description 66
objectives 66
payment methods for shop, example 67
- FileStrategy type** 167
- Finite State Machines (FSM)** 192, 239
- First In First Out (FIFO)** 123
- flow control**
about 18
if/else statement 18
switch statement 19
- Flyweight design pattern**
about 155
acceptance criteria 156
description 155
example 156
implementation 159, 160, 161, 163
objectives 155
structs 156, 158, 159
unit testing 156, 158, 159
versus Singleton design pattern 163
- functions**
about 20
anonymous function 21
closures 22
composition 20
errors, creating 22
errors, handling 22
errors, returning 22
returned types, naming 24
with undetermined parameters 23
- Future design pattern**
about 321
acceptance criteria 323
asynchronous requester, developing 323
description 321
implementation 329, 330, 331
objectives 323
- Template pattern, using** 332
unit test, writing 324, 325, 326, 328
- G**
- Gcode** 228
- GitHub**
Go open source projects, contributing 48
- Go get tool** 41
- Go-plus** 14
- Go**
advanced installation, on Linux 10
history 9
installing 9
installing, on Linux 10
installing, on Mac OS X 11
installing, on Windows 11
structural typing 145
URL 10, 12
workspace, setting 12
- godoc tool** 47
- gofmt tool** 46
- goimport tool** 47
- golang webpage** 10
- golint tool**
about 45
URL 45
- GOPATH** 9, 12
- Goroutines**
about 279
anonymous functions, launching 281, 282
Hello World program, creating 279, 281
WaitGroups 282, 283
- H**
- Hello World example**
creating 13
- HTTP Basic Authentication** 141
- HTTP REST API** 147
- I**
- if/else statement 18
- Inferred types 16
- inheritance
versus Composite design pattern 98
- Integrated Development Environment (IDE) 14

I
IntelliJ Gogland 14
Interface Definition Languages (IDL) 228
interfaces
 about 32
 contract, signing 32, 34
International Organization for Standardization (ISO) 148
Interpreter design pattern
 about 227
 acceptance criteria, for calculator 229
 advantages 238
 complexity 234
 example 228
 implementation 230, 231, 233, 234
 Interpreter interface, using 235, 236, 237
 objectives 228
 unit testing 229, 230

J

JavaScript Object Notation (JSON) 42
JSON data
 encoding package 43, 45
 managing 42, 43

L

libraries
 about 38, 39
 naming conventions 41
Linux
 Go, installing 10
LiteIDE 14

M

Mac OS X
 Go, installing 11
maps 24, 26
Mediator design pattern
 about 259
 acceptance criteria 260
 calculator, creating 260
 description 259
 implementation 260, 261, 263, 264
 objectives 259
 types, uncoupling 264
Memento design pattern

about 216
acceptance criteria 217
Care Taker 216
description 216
example 223, 226, 227
example, with strings 217
implementing 221, 223
Memento 216
objectives 216
Originator 216
requisites 217
summarizing 227
unit testing 217, 219, 221
minimal-mesos-go-framework 12
mutexes
 about 288
 concurrent counter example 289, 290
 race detector, using 290, 291, 293

O

Observer design pattern
 about 264
 acceptance criteria 265
 description 264
 implementation 270, 273
 notifier 265
 objectives 265
 unit tests, performing 266, 267, 268, 270
OpenCV 168
OpenWeatherMap API
 URL 148, 153
operators 17

P

parallelism
 versus concurrency 276
payment methods for shop example
 about 67
 acceptance criteria 67
 Debit card method, upgrading to new platform 73
 implementation 70
 unit test 67
Pipeline design pattern
 about 332
 acceptance criteria 333

concurrent multi-operation 333
description 332
implementation 335
objectives 332
testing 333, 334
usage 340
pointers
 about 29
 advantages 29
Promise 321
Prototype design pattern
 about 84, 90
 customized shirts shop, example 84
 description 84
 objective 84
Proxy design pattern
 about 122
 acceptance criteria 123
 around actions 131
 description 122
 example 123
 implementation 128, 131
 objectives 123
 unit testing 123, 124, 127, 128
 versus Decorator design pattern 146

R

race condition 288
race detector
 using 290, 291, 293
reminder keyword 216

S

Scala 276
Secure Shell (SSH) 51
select statement 298, 300
Singleton design pattern
 about 50, 56
 counter, example 51
 description 51
 objectives 51
 versus Flyweight design pattern 163
slices 24, 25
State design pattern
 about 252
acceptance criteria 253
description 252
game state, defining 258
game, building 259
implementation 253, 254, 256, 258
number game, creating 253
objectives 252
Stock Keeping Unit (SKU) 84
Strategy design pattern
 about 165
 acceptance criteria 167
 advantages 180
 description 166
 images, rendering 166
 implementation 168, 170, 173
 library issues, solving 173, 175, 176, 178, 180
 objectives 166
 text, rendering 166
structural typing 145
structure (structs) 29, 30, 32
switch statement 19

T

tandem bike 276
Template design pattern
 about 203
 acceptance criteria 205
 anonymous function, using 208, 211
 description 204
 example 204
 implementing 207, 208
 modifications, avoiding on interface 211, 213
 objectives 204
 requisites 205
 source code 214, 215
 summarizing 215
 unit testing 205, 207
Terminal program 13
Test Driven Development (TDD) 34, 37, 38
testing
 about 34
 package, using 35, 36
tools
 about 45
 godoc tool 47

gofmt tool 46
goimport tool 47
golint tool 45
transaction operation 227
types 15

U

unique counter example
about 51
acceptance criteria 52
implementation 55
requisites 52
unit tests, writing 52

V

variables 16
vehicle factory example
about 76
acceptance criteria 76
implementation 82
unit test 76
vehicle manufacturing example
about 57
acceptance criteria 58
manufacturing, implementation 62
requisites 58
unit test, for vehicle builder 58
visibility 26
Visitor design pattern

about 239
acceptance criteria 241
advantages 252
description 240
example 247, 248, 251
implementation 245, 247
log appender 240
objectives 240
unit tests, performing 241, 242, 244, 245

W

WaitGroups 282, 283
Windows
Go, installing 11
workers pool
acceptance criteria 344
description 343
Dispatcher interface, implementing 345, 346, 347
implementation 344, 345
objectives 343
pipeline 343
pipeline, developing 348, 349, 350, 351
testing 353, 355
usage 355
used, with app 351, 353

Z

zero-initialization 25, 27, 28

Module 3

Go Programming Blueprints, Second Edition

*Build real-world, production-ready solutions in Go using
cutting-edge technology and techniques*

1

Chat Application with Web Sockets

Go is great for writing high-performance, concurrent server applications and tools, and the Web is the perfect medium over which to deliver them. It would be difficult these days to find a gadget that is not web-enabled and this allows us to build a single application that targets almost all platforms and devices.

Our first project will be a web-based chat application that allows multiple users to have a real-time conversation right in their web browser. Idiomatic Go applications are often composed of many packages, which are organized by having code in different folders, and this is also true of the Go standard library. We will start by building a simple web server using the `net/http` package, which will serve the HTML files. We will then go on to add support for web sockets through which our messages will flow.

In languages such as C#, Java, or Node.js, complex threading code and clever use of locks need to be employed in order to keep all clients in sync. As we will see, Go helps us enormously with its built-in channels and concurrency paradigms.

In this chapter, you will learn how to:

- Use the `net/http` package to serve HTTP requests
- Deliver template-driven content to users' browsers
- Satisfy a Go interface to build our own `http.Handler` types
- Use Go's goroutines to allow an application to perform multiple tasks concurrently
- Use channels to share information between running goroutines
- Upgrade HTTP requests to use modern features such as web sockets
- Add tracing to the application to better understand its inner working

- Write a complete Go package using test-driven development practices
- Return unexported types through exported interfaces



Complete source code for this project can be found at <https://github.com/matryer/goblueprints/tree/master/chapter1/chat>. The source code was periodically committed so the history in GitHub actually follows the flow of this chapter too.

A simple web server

The first thing our chat application needs is a web server that has two main responsibilities:

- Serving the HTML and JavaScript chat clients that run in the user's browser
- Accepting web socket connections to allow the clients to communicate



The GOPATH environment variable is covered in detail in [Appendix, Good Practices for a Stable Go environment](#). Be sure to read that first if you need help getting set up.

Create a `main.go` file inside a new folder called `chat` in your GOPATH and add the following code:

```
package main
import (
    "log"
    "net/http"
)
func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte(`))
        <html>
            <head>
                <title>Chat</title>
            </head>
            <body>
                Let's chat!
            </body>
        </html>
    ))
})
// start the web server
if err := http.ListenAndServe(":8080", nil); err != nil {
```

```
    log.Fatal("ListenAndServe:", err)
}
}
```

This is a complete, albeit simple, Go program that will:

- Listen to the root path using the `net/http` package
- Write out the hardcoded HTML when a request is made
- Start a web server on port :8080 using the `ListenAndServe` method

The `http.HandleFunc` function maps the path pattern `/` to the function we pass as the second argument, so when the user hits `http://localhost:8080/`, the function will be executed. The function signature of `func(w http.ResponseWriter, r *http.Request)` is a common way of handling HTTP requests throughout the Go standard library.



We are using package `main` because we want to build and run our program from the command line. However, if we were building a reusable chatting package, we might choose to use something different, such as package `chat`.

In a terminal, run the program by navigating to the `main.go` file you just created and execute the following command:

```
go run main.go
```



The `go run` command is a helpful shortcut for running simple Go programs. It builds and executes a binary in one go. In the real world, you usually use `go build` yourself to create and distribute binaries. We will explore this later.

Open the browser and type `http://localhost:8080` to see the **Let's chat!** message.

Having the HTML code embedded within our Go code like this works, but it is pretty ugly and will only get worse as our projects grow. Next, we will see how templates can help us clean this up.

Separating views from logic using templates

Templates allow us to blend generic text with specific text, for instance, injecting a user's name into a welcome message. For example, consider the following template:

```
Hello {{name}}, how are you?
```

We are able to replace the `{{name}}` text in the preceding template with the real name of a person. So if Bruce signs in, he might see:

```
Hello Bruce, how are you?
```

The Go standard library has two main template packages: one called `text/template` for text and one called `html/template` for HTML. The `html/template` package does the same as the text version except that it understands the context in which data will be injected into the template. This is useful because it avoids script injection attacks and resolves common issues such as having to encode special characters for URLs.

Initially, we just want to move the HTML code from inside our Go code to its own file, but won't blend any text just yet. The template packages make loading external files very easy, so it's a good choice for us.

Create a new folder under our `chat` folder called `templates` and create a `chat.html` file inside it. We will move the HTML from `main.go` to this file, but we will make a minor change to ensure our changes have taken effect:

```
<html>
  <head>
    <title>Chat</title>
  </head>
  <body>
    Let's chat (from template)
  </body>
</html>
```

Now, we have our external HTML file ready to go, but we need a way to compile the template and serve it to the user's browser.



Compiling a template is a process by which the source template is interpreted and prepared for blending with various data, which must happen before a template can be used but only needs to happen once.

We are going to write our own `struct` type that is responsible for loading, compiling, and delivering our template. We will define a new type that will take a `filename` string, compile the template once (using the `sync.Once` type), keep the reference to the compiled template, and then respond to HTTP requests. You will need to import the `text/template`, `path/filepath`, and `sync` packages in order to build your code.

In `main.go`, insert the following code above the `func main()` line:

```
// templ represents a single template
type templateHandler struct {
    once      sync.Once
    filename string
    templ     *template.Template
}
// ServeHTTP handles the HTTP request.
func (t *templateHandler) ServeHTTP(w http.ResponseWriter, r
*http.Request) {
    t.once.Do(func() {
        t.templ = template.Must(template.ParseFiles(filepath.Join("templates",
        t.filename)))
    })
    t.templ.Execute(w, nil)
}
```



Did you know that you could automate the adding and removing of imported packages? See Appendix, *Good Practices for a Stable Go Environment*, on how to do this.

The `templateHandler` type has a single method called `ServeHTTP` whose signature looks suspiciously like the method we passed to `http.HandleFunc` earlier. This method will load the source file, compile the template and execute it, and write the output to the specified `http.ResponseWriter` method. Because the `ServeHTTP` method satisfies the `http.Handler` interface, we can actually pass it directly to `http.Handle`.



A quick look at the Go standard library source code, which is located at [ht tp://golang.org/pkg/net/http/#Handler](https://golang.org/pkg/net/http/#Handler), will reveal that the interface definition for `http.Handler` specifies that only the `ServeHTTP` method need be present in order for a type to be used to serve HTTP requests by the `net/http` package.

Doing things once

We only need to compile the template once, and there are a few different ways to approach this in Go. The most obvious is to have a `NewTemplateHandler` function that creates the type and calls some initialization code to compile the template. If we were sure the function would be called by only one goroutine (probably the main one during the setup in the `main` function), this would be a perfectly acceptable approach. An alternative, which we have employed in the preceding section, is to compile the template once inside the `ServeHTTP` method. The `sync.Once` type guarantees that the function we pass as an argument will only be executed once, regardless of how many goroutines are calling `ServeHTTP`. This is helpful because web servers in Go are automatically concurrent and once our chat application takes the world by storm, we could very well expect to have many concurrent calls to the `ServeHTTP` method.

Compiling the template inside the `ServeHTTP` method also ensures that our code does not waste time doing work before it is definitely needed. This lazy initialization approach doesn't save us much in our present case, but in cases where the setup tasks are time- and resource-intensive and where the functionality is used less frequently, it's easy to see how this approach would come in handy.

Using your own handlers

To implement our `templateHandler` type, we need to update the `main body` function so that it looks like this:

```
func main() {
    // root
    http.Handle("/", &templateHandler{filename: "chat.html"})
    // start the web server
    if err := http.ListenAndServe(":8080", nil); err != nil {
        log.Fatal("ListenAndServe:", err)
    }
}
```

The `templateHandler` structure is a valid `http.Handler` type so we can pass it directly to the `http.Handle` function and ask it to handle requests that match the specified pattern. In the preceding code, we created a new object of the type `templateHandler`, specifying the `filename` as `chat.html` that we then take the address of (using the `&` address of the operator) and pass it to the `http.Handle` function. We do not store a reference to our newly created `templateHandler` type, but that's OK because we don't need to refer to it again.

In your terminal, exit the program by pressing *Ctrl + C* and re-run it, then refresh your browser and notice the addition of the (from template) text. Now our code is much simpler than an HTML code and free from its ugly blocks.

Properly building and executing Go programs

Running Go programs using a `go run` command is great when our code is made up of a single `main.go` file. However, often we might quickly need to add other files. This requires us to properly build the whole package into an executable binary before running it. This is simple enough, and from now on, this is how you will build and run your programs in a terminal:

```
go build -o {name}  
.{name}
```

The `go build` command creates the output binary using all the `.go` files in the specified folder, and the `-o` flag indicates the name of the generated binary. You can then just run the program directly by calling it by name.

For example, in the case of our chat application, we could run:

```
go build -o chat  
./chat
```

Since we are compiling templates the first time the page is served, we will need to restart your web server program every time anything changes in order to see the changes take effect.

Modeling a chat room and clients on the server

All users (clients) of our chat application will automatically be placed in one big public room where everyone can chat with everyone else. The `room` type will be responsible for managing client connections and routing messages in and out, while the `client` type represents the connection to a single client.

Go refers to classes as types and instances of those classes as objects.



To manage our web sockets, we are going to use one of the most powerful aspects of the Go community open source third-party packages. Every day, new packages solving real-world problems are released, ready for you to use in your own projects, and they even allow you to add features, report and fix bugs, and get support.



It is often unwise to reinvent the wheel unless you have a very good reason. So before embarking on building a new package, it is worth searching for any existing projects that might have already solved your very problem. If you find one similar project that doesn't quite satisfy your needs, consider contributing to the project and adding features. Go has a particularly active open source community (remember that Go itself is open source) that is always ready to welcome new faces or avatars.

We are going to use Gorilla Project's `websocket` package to handle our server-side sockets rather than write our own. If you're curious about how it works, head over to the project home page on GitHub, <https://github.com/gorilla/websocket>, and browse the open source code.

Modeling the client

Create a new file called `client.go` alongside `main.go` in the `chat` folder and add the following code:

```
package main
import (
    "github.com/gorilla/websocket"
)
// client represents a single chatting user.
type client struct {
    // socket is the web socket for this client.
    socket *websocket.Conn
    // send is a channel on which messages are sent.
    send chan []byte
    // room is the room this client is chatting in.
    room *room
}
```

In the preceding code, `socket` will hold a reference to the web socket that will allow us to communicate with the client, and the `send` field is a buffered channel through which received messages are queued ready to be forwarded to the user's browser (via the `socket`). The `room` field will keep a reference to the room that the client is chatting in this is required so that we can forward messages to everyone else in the room.

If you try to build this code, you will notice a few errors. You must ensure that you have called `go get` to retrieve the `websocket` package, which is as easy as opening a terminal and typing the following:

```
go get github.com/gorilla/websocket
```

Building the code again will yield another error:

```
./client.go:17 undefined: room
```

The problem is that we have referred to a `room` type without defining it anywhere. To make the compiler happy, create a file called `room.go` and insert the following placeholder code:

```
package main
type room struct {
    // forward is a channel that holds incoming messages
    // that should be forwarded to the other clients.
    forward chan []byte
}
```

We will improve this definition later once we know a little more about what our room needs to do, but for now, this will allow us to proceed. Later, the `forward` channel is what we will use to send the incoming messages to all other clients.



You can think of channels as an in-memory thread-safe message queue where senders pass data and receivers read data in a non-blocking, thread-safe way.

In order for a client to do any work, we must define some methods that will do the actual reading and writing to and from the web socket. Adding the following code to `client.go` outside (underneath) the `client` struct will add two methods called `read` and `write` to the `client` type:

```
func (c *client) read() {
    defer c.socket.Close()
    for {
        _, msg, err := c.socket.ReadMessage()
        if err != nil {
```

```
        return
    }
    c.room.forward <- msg
}
}

func (c *client) write() {
    defer c.socket.Close()
    for msg := range c.send {
        err := c.socket.WriteMessage(websocket.TextMessage, msg)
        if err != nil {
            return
        }
    }
}
```

The `read` method allows our client to read from the socket via the `ReadMessage` method, continually sending any received messages to the `forward` channel on the `room` type. If it encounters an error (such as 'the socket has died'), the loop will break and the socket will be closed. Similarly, the `write` method continually accepts messages from the `send` channel writing everything out of the socket via the `WriteMessage` method. If writing to the socket fails, the `for` loop is broken and the socket is closed. Build the package again to ensure everything compiles.



In the preceding code, we introduced the `defer` keyword, which is worth exploring a little. We are asking Go to run `c.socket.Close()` when the function exits. It's extremely useful for when you need to do some tidying up in a function (such as closing a file or, as in our case, a socket) but aren't sure where the function will exit. As our code grows, if this function has multiple `return` statements, we won't need to add any more calls to close the socket, because this single `defer` statement will catch them all. Some people complain about the performance of using the `defer` keyword, since it doesn't perform as well as typing the `close` statement before every exit point in the function. You must weigh up the runtime performance cost against the code maintenance cost and potential bugs that may get introduced if you decide not to use `defer`. As a general rule of thumb, writing clean and clear code wins; after all, we can always come back and optimize any bits of code we feel is slowing our product down if we are lucky enough to have such success.

Modeling a room

We need a way for clients to join and leave rooms in order to ensure that the `c.room.forward <- msg` code in the preceding section actually forwards the message to all the clients. To ensure that we are not trying to access the same data at the same time, a sensible approach is to use two channels: one that will add a client to the room and another that will remove it. Let's update our `room.go` code to look like this:

```
package main
type room struct {
    // forward is a channel that holds incoming messages
    // that should be forwarded to the other clients.
    forward chan []byte
    // join is a channel for clients wishing to join the room.
    join chan *client
    // leave is a channel for clients wishing to leave the room.
    leave chan *client
    // clients holds all current clients in this room.
    clients map[*client]bool
}
```

We have added three fields: two channels and a map. The `join` and `leave` channels exist simply to allow us to safely add and remove clients from the `clients` map. If we were to access the map directly, it is possible that two goroutines running concurrently might try to modify the map at the same time, resulting in corrupt memory or unpredictable state.

Concurrency programming using idiomatic Go

Now we get to use an extremely powerful feature of Go's concurrency offerings the `select` statement. We can use `select` statements whenever we need to synchronize or modify shared memory, or take different actions depending on the various activities within our channels.

Beneath the `room` structure, add the following `run` method that contains three `select` cases:

```
func (r *room) run() {
    for {
        select {
        case client := <-r.join:
            // joining
            r.clients[client] = true
        case client := <-r.leave:
            // leaving
        }
    }
}
```

```
    delete(r.clients, client)
    close(client.send)
  case msg := <-r.forward:
    // forward message to all clients
    for client := range r.clients {
      client.send <- msg
    }
}
}
```

Although this might seem like a lot of code to digest, once we break it down a little, we will see that it is fairly simple, although extremely powerful. The top `for` loop indicates that this method will run forever, until the program is terminated. This might seem like a mistake, but remember, if we run this code as a goroutine, it will run in the background, which won't block the rest of our application. The preceding code will keep watching the three channels inside our room: `join`, `leave`, and `forward`. If a message is received on any of those channels, the `select` statement will run the code for that particular case.



It is important to remember that it will only run one block of case code at a time. This is how we are able to synchronize to ensure that our `r.clients` map is only ever modified by one thing at a time.

If we receive a message on the `join` channel, we simply update the `r.clients` map to keep a reference of the client that has joined the room. Notice that we are setting the value to `true`. We are using the map more like a slice, but do not have to worry about shrinking the slice as clients come and go through time setting the value to `true` is just a handy, low-memory way of storing the reference.

If we receive a message on the `leave` channel, we simply delete the `client` type from the map, and close its `send` channel. If we receive a message on the `forward` channel, we iterate over all the clients and add the message to each client's `send` channel. Then, the `write` method of our client type will pick it up and send it down the socket to the browser.

Turning a room into an HTTP handler

Now we are going to turn our `room` type into an `http.Handler` type like we did with the template handler earlier. As you will recall, to do this, we must simply add a method called `ServeHTTP` with the appropriate signature.

Add the following code to the bottom of the `room.go` file:

```
const (
    socketBufferSize = 1024
    messageBufferSize = 256
)
var upgrader = &websocket.Upgrader{ReadBufferSize: socketBufferSize,
    WriteBufferSize: socketBufferSize}
func (r *room) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    socket, err := upgrader.Upgrade(w, req, nil)
    if err != nil {
        log.Fatal("ServeHTTP:", err)
        return
    }
    client := &client{
        socket: socket,
        send:   make(chan []byte, messageBufferSize),
        room:   r,
    }
    r.join <- client
    defer func() { r.leave <- client }()
    go client.write()
    client.read()
}
```

The `ServeHTTP` method means a room can now act as a handler. We will implement it shortly, but first let's have a look at what is going on in this snippet of code.



If you accessed the chat endpoint in a web browser, you would likely crash the program and see an error like **ServeHTTPwebsocket: version != 13**. This is because it is intended to be accessed via a web socket rather than a web browser.

In order to use web sockets, we must upgrade the HTTP connection using the `websocket.Upgrader` type, which is reusable so we need only create one. Then, when a request comes in via the `ServeHTTP` method, we get the socket by calling the `upgrader.Upgrade` method. All being well, we then create our client and pass it into the `join` channel for the current room. We also defer the leaving operation for when the client is finished, which will ensure everything is tidied up after a user goes away.

The `write` method for the client is then called as a goroutine, as indicated by the three characters at the beginning of the line `go` (the word `go` followed by a space character). This tells Go to run the method in a different thread or goroutine.



Compare the amount of code needed to achieve multithreading or concurrency in other languages with the three key presses that achieve it in Go, and you will see why it has become a favorite among system developers.

Finally, we call the `read` method in the main thread, which will block operations (keeping the connection alive) until it's time to close it. Adding constants at the top of the snippet is a good practice for declaring values that would otherwise be hardcoded throughout the project. As these grow in number, you might consider putting them in a file of their own, or at least at the top of their respective files so they remain easy to read and modify.

Using helper functions to remove complexity

Our room is almost ready to go, although in order for it to be of any use, the channels and map need to be created. As it is, this could be achieved by asking the developer to use the following code to be sure to do this:

```
r := &room{
    forward: make(chan []byte),
    join:    make(chan *client),
    leave:   make(chan *client),
    clients: make(map[*client]bool),
}
```

Another, slightly more elegant, solution is to instead provide a `newRoom` function that does this for us. This removes the need for others to know about exactly what needs to be done in order for our room to be useful. Underneath the `type room struct` definition, add this function:

```
// newRoom makes a new room.
func newRoom() *room {
    return &room{
        forward: make(chan []byte),
        join:    make(chan *client),
        leave:   make(chan *client),
        clients: make(map[*client]bool),
    }
}
```

Now the users of our code need only call the `newRoom` function instead of the more verbose six lines of code.

Creating and using rooms

Let's update our main function in `main.go` to first create and then run a room for everybody to connect to:

```
func main() {
    r := newRoom()
    http.Handle("/", &templateHandler{filename: "chat.html"})
    http.Handle("/room", r)
    // get the room going
    go r.run()
    // start the web server
    if err := http.ListenAndServe(":8080", nil); err != nil {
        log.Fatal("ListenAndServe:", err)
    }
}
```

We are running the room in a separate goroutine (notice the `go` keyword again) so that the chatting operations occur in the background, allowing our main goroutine to run the web server. Our server is now finished and successfully built, but remains useless without clients to interact with.

Building an HTML and JavaScript chat client

In order for the users of our chat application to interact with the server and therefore other users, we need to write some client-side code that makes use of the web sockets found in modern browsers. We are already delivering HTML content via the template when users hit the root of our application, so we can enhance that.

Update the `chat.html` file in the `templates` folder with the following markup:

```
<html>
    <head>
        <title>Chat</title>
        <style>
            input { display: block; }
            ul { list-style: none; }
        </style>
    </head>
    <body>
        <ul id="messages"></ul>
        <form id="chatbox">
            <textarea></textarea>
            <input type="submit" value="Send" />
```

```
</form>  </body>  
</html>
```

The preceding HTML will render a simple web form on the page containing a text area and a **Send** button this is how our users will submit messages to the server. The `messages` element in the preceding code will contain the text of the chat messages so that all the users can see what is being said. Next, we need to add some JavaScript to add some functionality to our page. Underneath the `form` tag, above the closing `</body>` tag, insert the following code:

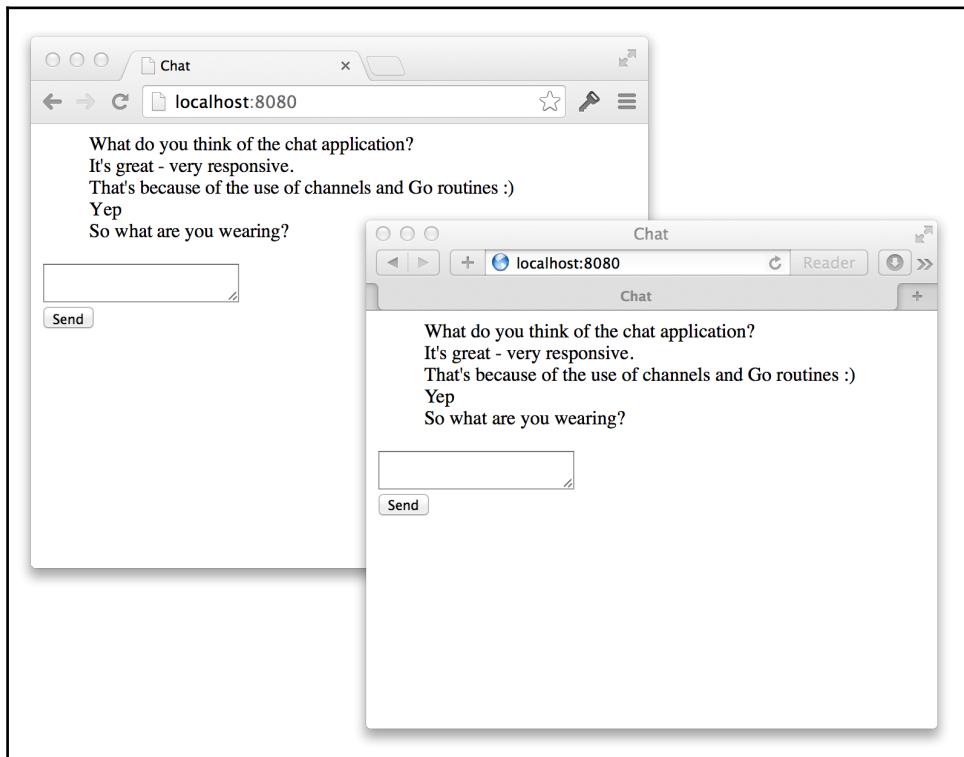
```
<script  src="//ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js">  
</script>  
<script>  
$(function() {  
    var socket = null;  
    var msgBox = $("#chatbox textarea");  
    var messages = $("#messages");  
    $("#chatbox").submit(function(){  
        if (!msgBox.val()) return false;  
        if (!socket) {  
            alert("Error: There is no socket connection.");  
            return false;  
        }  
        socket.send(msgBox.val());  
        msgBox.val("");  
        return false;  
    });  
    if (!window["WebSocket"]) {  
        alert("Error: Your browser does not support web sockets.");  
    } else {  
        socket = new WebSocket("ws://localhost:8080/room");  
        socket.onclose = function() {  
            alert("Connection has been closed.");  
        }  
        socket.onmessage = function(e) {  
            messages.append($(".list-item").text(e.data));  
        }  
    }  
});  
</script>
```

The `socket = new WebSocket("ws://localhost:8080/room")` line is where we open the socket and add event handlers for two key events: `onclose` and `onmessage`. When the socket receives a message, we use jQuery to append the message to the list element and thus present it to the user.

Submitting the HTML form triggers a call to `socket.send`, which is how we send messages to the server.

Build and run the program again to ensure the templates recompile so these changes are represented.

Navigate to `http://localhost:8080/` in two separate browsers (or two tabs of the same browser) and play with the application. You will notice that messages sent from one client appear instantly in the other clients:



Getting more out of templates

Currently, we are using templates to deliver static HTML, which is nice because it gives us a clean and simple way to separate the client code from the server code. However, templates are actually much more powerful, and we are going to tweak our application to make some more realistic use of them.

The host address of our application (`:8080`) is hardcoded at two places at the moment. The first instance is in `main.go` where we start the web server:

```
if err := http.ListenAndServe(":8080", nil); err != nil {  
    log.Fatal("ListenAndServe:", err)  
}
```

The second time it is hardcoded in the JavaScript when we open the socket:

```
socket = new WebSocket("ws://localhost:8080/room");
```

Our chat application is pretty stubborn if it insists on only running locally on port 8080, so we are going to use command-line flags to make it configurable and then use the injection capabilities of templates to make sure our JavaScript knows the right host.

Update your `main` function in `main.go`:

```
func main() {  
    var addr = flag.String("addr", ":8080", "The addr of the application.")  
    flag.Parse() // parse the flags  
    r := newRoom()  
    http.Handle("/", &templateHandler{filename: "chat.html"})  
    http.Handle("/room", r)  
    // get the room going  
    go r.run()  
    // start the web server  
    log.Println("Starting web server on", *addr)  
    if err := http.ListenAndServe(*addr, nil); err != nil {  
        log.Fatal("ListenAndServe:", err)  
    }  
}
```

You will need to import the `flag` package in order for this code to build. The definition for the `addr` variable sets up our flag as a string that defaults to `:8080` (with a short description of what the value is intended for). We must call `flag.Parse()` that parses the arguments and extracts the appropriate information. Then, we can reference the value of the host flag by using `*addr`.



The call to `flag.String` returns a type of `*string`, which is to say it returns the address of a string variable where the value of the flag is stored. To get the value itself (and not the address of the value), we must use the pointer indirection operator, `*`.

We also added a `log.Println` call to output the address in the terminal so we can be sure that our changes have taken effect.

We are going to modify the `templateHandler` type we wrote so that it passes the details of the request as data into the template's `Execute` method. In `main.go`, update the `ServeHTTP` function to pass the request `r` as the `data` argument to the `Execute` method:

```
func (t *templateHandler) ServeHTTP(w http.ResponseWriter, r
    *http.Request) {
    t.once.Do(func() {
        t.templ = template.Must(template.ParseFiles(filepath.Join("templates",
            t.filename)))
    })
    t.templ.Execute(w, r)
}
```

This tells the template to render itself using data that can be extracted from `http.Request`, which happens to include the host address that we need.

To use the `Host` value of `http.Request`, we can then make use of the special template syntax that allows us to inject data. Update the line where we create our socket in the `chat.html` file:

```
socket = new WebSocket("ws://{{.Host}}/room");
```

The double curly braces represent an annotation and the way we tell our template source to inject data. The `{{.Host}}` is essentially equivalent of telling it to replace the annotation with the value from `request.Host` (since we passed the `request.r` object in as data).



We have only scratched the surface of the power of the templates built into Go's standard library. The `text/template` package documentation is a great place to learn more about what you can achieve. You can find more about it at <http://golang.org/pkg/text/template>.

Rebuild and run the chat program again, but this time notice that the chatting operations no longer produce an error, whichever host we specify:

```
go build -o chat
./chat -addr=:3000"
```

View the source of the page in the browser and notice that `{{.Host}}` has been replaced with the actual host of the application. Valid hosts aren't just port numbers; you can also specify the IP addresses or other hostnames provided they are allowed in your environment, for example, `-addr="192.168.0.1:3000"`.

Tracing code to get a look under the hood

The only way we will know that our application is working is by opening two or more browsers and using our UI to send messages. In other words, we are manually testing our code. This is fine for experimental projects such as our chat application or small projects that aren't expected to grow, but if our code is to have a longer life or be worked on by more than one person, manual testing of this kind becomes a liability. We are not going to tackle **Test-driven Development (TDD)** for our chat program, but we should explore another useful debugging technique called tracing.

Tracing is a practice by which we log or print key steps in the flow of a program to make what is going on under the covers visible. In the previous section, we added a `log.Println` call to output the address that the chat program was binding to. In this section, we are going to formalize this and write our own complete tracing package.

We are going to explore TDD practices when writing our tracing code because TDD is a perfect example of a package that we are likely to reuse, add to, share, and hopefully, even open source.

Writing a package using TDD

Packages in Go are organized into folders, with one package per folder. It is a build error to have differing package declarations within the same folder because all sibling files are expected to contribute to a single package. Go has no concept of subpackages, which means nested packages (in nested folders) exist only for aesthetic or informational reasons but do not inherit any functionality or visibility from super packages. In our chat application, all of our files contributed to the `main` package because we wanted to build an executable tool. Our tracing package will never be run directly, so it can and should use a different package name. We will also need to think about the **Application Programming Interface (API)** of our package, considering how to model a package so that it remains as extensible and flexible as possible for users. This includes the fields, functions, methods, and types that should be exported (visible to the user) and remain hidden for simplicity's sake.



Go uses capitalization of names to denote which items are exported such that names that begin with a capital letter (for example, `Tracer`) are visible to users of a package, and names that begin with a lowercase letter (for example, `templateHandler`) are hidden or private.

Create a new folder called `trace`, which will be the name of our tracing package, alongside the `chat` folder so that the folder structure now looks like this:

```
/chat
    client.go
    main.go
    room.go
/trace
```

Before we jump into code, let's agree on some design goals for our package by which we can measure success:

- The package should be easy to use
- Unit tests should cover the functionality
- Users should have the flexibility to replace the tracer with their own implementation

Interfaces

Interfaces in Go are an extremely powerful language feature that allows us to define an API without being strict or specific on the implementation details. Wherever possible, describing the basic building blocks of your packages using interfaces usually ends up paying dividends down the road, and this is where we will start for our tracing package.

Create a new file called `tracer.go` inside the `trace` folder and write the following code:

```
package trace
// Tracer is the interface that describes an object capable of
// tracing events throughout code.
type Tracer interface {
    Trace(...interface{})
}
```

The first thing to notice is that we have defined our package as `trace`.



While it is a good practice to have the folder name match the package name, Go tools do not enforce it, which means you are free to name them differently if it makes sense. Remember, when people import your package, they will type the name of the folder, and if suddenly a package with a different name is imported, it could get confusing.

Our `Tracer` type (the capital `T` means we intend this to be a publicly visible type) is an interface that describes a single method called `Trace`. The `...interface{}` argument type states that our `Trace` method will accept zero or more arguments of any type. You might think that this is a redundant provision as the method should just take a single string (we want to just trace out some string of characters, don't we?). However, consider functions such as `fmt.Sprint` and `log.Fatal`, both of which follow a pattern littered throughout Go's standard library that provides a helpful shortcut when trying to communicate multiple things in one go. Wherever possible, we should follow such patterns and practices because we want our own APIs to be familiar and clear to the Go community.

Unit tests

We promised ourselves that we would follow test-driven practices, but interfaces are simply definitions that do not provide any implementation and so cannot be directly tested. But we are about to write a real implementation of a `Tracer` method, and we will indeed write the tests first.

Create a new file called `tracer_test.go` in the `trace` folder and insert the following scaffold code:

```
package trace
import (
    "testing"
)
func TestNew(t *testing.T) {
    t.Error("We haven't written our test yet")
}
```

Testing was built into the Go tool chain from the very beginning, making writing automatable tests a first-class citizen. The test code lives alongside the production code in files suffixed with `_test.go`. The Go tools will treat any function that starts with `Test` (taking a single `*testing.T` argument) as a unit test, and it will be executed when we run our tests. To run them for this package, navigate to the `trace` folder in a terminal and do the following:

```
go test
```

You will see that our tests fail because of our call to `t.Error` in the body of our `TestNew` function:

```
--- FAIL: TestNew (0.00 seconds)
tracer_test.go:8: We haven't written our test yet
FAIL
exit status 1
FAIL trace 0.011s
```



Clearing the terminal before each test run is a great way to make sure you aren't confusing previous runs with the most recent one. On Windows, you can use the `cls` command; on Unix machines, the `clear` command does the same thing.

Obviously, we haven't properly written our test and we don't expect it to pass yet, so let's update the `TestNew` function:

```
func TestNew(t *testing.T) {
    var buf bytes.Buffer
    tracer := New(&buf)
    if tracer == nil {
        t.Error("Return from New should not be nil")
    } else {
        tracer.Trace("Hello trace package.")
        if buf.String() != "Hello trace package.\n" {
            t.Errorf("Trace should not write '%s'.", buf.String())
        }
    }
}
```

Most packages throughout the book are available from the Go standard library, so you can add an `import` statement for the appropriate package in order to access the package. Others are external, and that's when you need to use `go get` to download them before they can be imported. For this case, you'll need to add `import "bytes"` to the top of the file.

We have started designing our API by becoming the first user of it. We want to be able to capture the output of our tracer in a `bytes.Buffer` variable so that we can then ensure that the string in the buffer matches the expected value. If it does not, a call to `t.Errorf` will fail the test. Before that, we check to make sure the return from a made-up `New` function is not `nil`; again, if it is, the test will fail because of the call to `t.Error`.

Red-green testing

Running `go test` now actually produces an error; it complains that there is no `New` function. We haven't made a mistake here; we are following a practice known as red-green testing. Red-green testing proposes that we first write a unit test, see it fail (or produce an error), write the minimum amount of code possible to make that test pass, and rinse and repeat it again. The key point here being that we want to make sure the code we add is actually doing something as well as ensuring that the test code we write is testing something meaningful.

Consider a meaningless test for a minute:

```
if true == true {  
    t.Error("True should be true")  
}
```

It is logically impossible for `true` to not be `true` (if `true` ever equals `false`, it's time to get a new computer), and so our test is pointless. If a test or claim cannot fail, there is no value whatsoever to be found in it. Replacing `true` with a variable that you expect to be set to `true` under certain conditions would mean that such a test can indeed fail (like when the code being tested is misbehaving) at this point, you have a meaningful test that is worth contributing to the code base.

You can treat the output of `go test` like a to-do list, solving only one problem at a time. Right now, the complaint about the missing `New` function is all we will address. In the `tracer.go` file, let's add the minimum amount of code possible to progress with things; add the following snippet underneath the interface type definition:

```
func New() {}
```

Running `go test` now shows us that things have indeed progressed, albeit not very far. We now have two errors:

```
./tracer_test.go:11: too many arguments in call to New  
./tracer_test.go:11: New(&buf) used as value
```

The first error tells us that we are passing arguments to our `New` function, but the `New` function doesn't accept any. The second error says that we are using the return of the `New` function as a value, but that the `New` function doesn't return anything. You might have seen this coming, and indeed as you gain more experience writing test-driven code, you will most likely jump over such trivial details. However, to properly illustrate the method, we are going to be pedantic for a while. Let's address the first error by updating our `New` function to take in the expected argument:

```
func New(w io.Writer) {}
```

We are taking an argument that satisfies the `io.Writer` interface, which means that the specified object must have a suitable `Write` method.



Using existing interfaces, especially ones found in the Go standard library, is an extremely powerful and often necessary way to ensure that your code is as flexible and elegant as possible.

Accepting `io.Writer` means that the user can decide where the tracing output will be written. This output could be the standard output, a file, network socket, `bytes.Buffer` as in our test case, or even some custom-made object, provided it can act like an `io.Writer` interface.

Running `go test` again shows us that we have resolved the first error and we only need add a return type in order to progress past our second error:

```
func New(w io.Writer) Tracer {}
```

We are stating that our `New` function will return a `Tracer`, but we do not return anything, which `go test` happily complains about:

```
./tracer.go:13: missing return at end of function
```

Fixing this is easy; we can just return `nil` from the `New` function:

```
func New(w io.Writer) Tracer {
    return nil
}
```

Of course, our test code has asserted that the return should not be `nil`, so `go test` now gives us a failure message:

```
tracer_test.go:14: Return from New should not be nil
```

You can see how this hyper-strict adherence to the red-green principle can get a little tedious, but it is vital that we do not jump too far ahead. If we were to write a lot of implementation code in one go, we will very likely have code that is not covered by a unit test. The ever-thoughtful core team has even solved this problem for us by providing code coverage statistics. The following command provides code statistics:

```
go test -cover
```

Provided that all tests pass, adding the `-cover` flag will tell us how much of our code was touched during the execution of the tests. Obviously, the closer we get to 100 percent the better.

Implementing the interface

To satisfy this test, we need something that we can properly return from the `New` method because `Tracer` is only an interface and we have to return something real. Let's add an implementation of a tracer to our `tracer.go` file:

```
type tracer struct {
    out io.Writer
}
func (t *tracer) Trace(a ...interface{}) {}
```

Our implementation is extremely simple: the `tracer` type has an `io.Writer` field called `out` which is where we will write the trace output to. And the `Trace` method exactly matches the method required by the `Tracer` interface, although it doesn't do anything yet.

Now we can finally fix the `New` method:

```
func New(w io.Writer) Tracer {
    return &tracer{out: w}
}
```

Running `go test` again shows us that our expectation was not met because nothing was written during our call to `Trace`:

```
tracer_test.go:18: Trace should not write ''.
```

Let's update our `Trace` method to write the blended arguments to the specified `io.Writer` field:

```
func (t *tracer) Trace(a ...interface{}) {
    fmt.Fprint(t.out, a...)
    fmt.Fprintln(t.out)
}
```

When the `Trace` method is called, we use `fmt.Fprint` (and `fmt.Fprintln`) to format and write the trace details to the `out` writer.

Have we finally satisfied our test?

```
go test -cover
PASS
coverage: 100.0% of statements
ok      trace 0.011s
```

Congratulations! We have successfully passed our test and have 100 percent test coverage. Once we have finished our glass of champagne, we can take a minute to consider something very interesting about our implementation.

Unexported types being returned to users

The `tracer` struct type we wrote is **unexported** because it begins with a lowercase `t`, so how is it that we are able to return it from the exported `New` function? After all, doesn't the user receive the returned object? This is perfectly acceptable and valid Go code; the user will only ever see an object that satisfies the `Tracer` interface and will never even know about our private `tracer` type. Since they only interact with the interface anyway, it wouldn't matter if our `tracer` implementation exposed other methods or fields; they would never be seen. This allows us to keep the public API of our package clean and simple.

This hidden implementation technique is used throughout the Go standard library; for example, the `ioutil.NopCloser` method is a function that turns a normal `io.Reader` interface into `io.ReadCloser` where the `Close` method does nothing (used for when `io.Reader` objects that don't need to be closed are passed into functions that require `io.ReadCloser` types). The method returns `io.ReadCloser` as far as the user is concerned, but under the hood, there is a secret `nopCloser` type hiding the implementation details.



To see this for yourself, browse the Go standard library source code at <http://golang.org/src/pkg/io/ioutil/ioutil.go> and search for the `nopCloser` struct.

Using our new trace package

Now that we have completed the first version of our `trace` package, we can use it in our chat application in order to better understand what is going on when users send messages through the user interface.

In `room.go`, let's import our new package and make some calls to the `Trace` method. The path to the `trace` package we just wrote will depend on your `GOPATH` environment variable because the import path is relative to the `$GOPATH/src` folder. So if you create your `trace` package in `$GOPATH/src/mycode/trace`, then you would need to import `mycode/trace`.

Update the `room` type and the `run()` method like this:

```
type room struct {
    // forward is a channel that holds incoming messages
    // that should be forwarded to the other clients.
    forward chan []byte
    // join is a channel for clients wishing to join the room.
    join chan *client
    // leave is a channel for clients wishing to leave the room.
    leave chan *client
    // clients holds all current clients in this room.
    clients map[*client]bool
    // tracer will receive trace information of activity
    // in the room.
    tracer trace.Tracer
}
func (r *room) run() {
    for {
        select {
        case client := <-r.join:
            // joining
            r.clients[client] = true
            r.tracer.Trace("New client joined")
        case client := <-r.leave:
            // leaving
            delete(r.clients, client)
            close(client.send)
            r.tracer.Trace("Client left")
        case msg := <-r.forward:
```

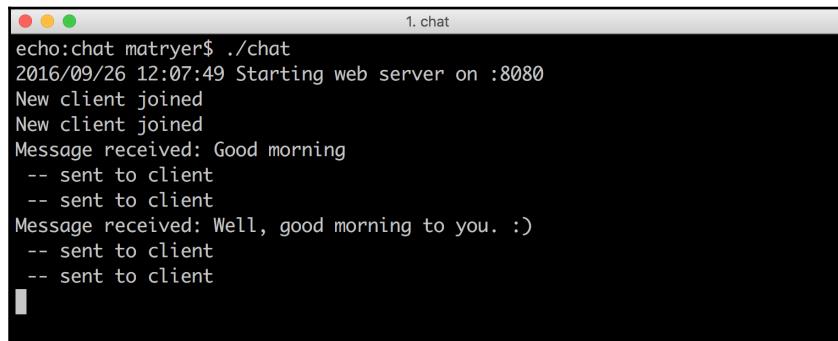
```
r.tracer.Trace("Message received: ", string(msg))
// forward message to all clients
for client := range r.clients {
    client.send <- msg
    r.tracer.Trace("-- sent to client")
}
}
}
}
```

We added a `tracer` field to our `room` type and then made periodic calls to the `Trace` method peppered throughout the code. If we run our program and try to send messages, you'll notice that the application panics because the `tracer` field is `nil`. We can remedy this for now by making sure we create and assign an appropriate object when we create our `room` type. Update the `main.go` file to do this:

```
r := newRoom()
r.tracer = trace.New(os.Stdout)
```

We are using our `New` method to create an object that will send the output to the `os.Stdout` standard output pipe (this is a technical way of saying we want it to print the output to our terminal).

Rebuild and run the program and use two browsers to play with the application, and notice that the terminal now has some interesting trace information for us:



The screenshot shows a terminal window titled "1. chat". The log output is as follows:

```
echo:chat matryer$ ./chat
2016/09/26 12:07:49 Starting web server on :8080
New client joined
New client joined
Message received: Good morning
-- sent to client
-- sent to client
Message received: Well, good morning to you. :)
-- sent to client
-- sent to client
```

Now we are able to use the debug information to get an insight into what the application is doing, which will assist us when developing and supporting our project.

Making tracing optional

Once the application is released, the sort of tracing information we are generating will be pretty useless if it's just printed out to some terminal somewhere, or even worse, if it creates a lot of noise for our system administrators. Also, remember that when we don't set a tracer for our `room` type, our code panics, which isn't a very user-friendly situation. To resolve these two issues, we are going to enhance our `trace` package with a `trace.Off()` method that will return an object that satisfies the `Tracer` interface but will not do anything when the `Trace` method is called.

Let's add a test that calls the `Off` function to get a silent tracer before making a call to `Trace` to ensure the code doesn't panic. Since the tracing won't happen, that's all we can do in our test code. Add the following test function to the `tracer_test.go` file:

```
func TestOff(t *testing.T) {
    var silentTracer Tracer = Off()
    silentTracer.Trace("something")
}
```

To make it pass, add the following code to the `tracer.go` file:

```
type nilTracer struct{}

func (t *nilTracer) Trace(a ...interface{}) {}

// Off creates a Tracer that will ignore calls to Trace.
func Off() Tracer {
    return &nilTracer{}
}
```

Our `nilTracer` struct has defined a `Trace` method that does nothing, and a call to the `Off()` method will create a new `nilTracer` struct and return it. Notice that our `nilTracer` struct differs from our `tracer` struct in that it doesn't take an `io.Writer` interface; it doesn't need one because it isn't going to write anything.

Now let's solve our second problem by updating our `newRoom` method in the `room.go` file:

```
func newRoom() *room {
    return &room{
        forward: make(chan []byte),
        join:     make(chan *client),
        leave:   make(chan *client),
        clients: map[*client]bool{},
        tracer:  trace.Off(),
    }
}
```

By default, our `room` type will be created with a `nilTracer` struct and any calls to `Trace` will just be ignored. You can try this out by removing the `r.tracer = trace.New(os.Stdout)` line from the `main.go` file: notice that nothing gets written to the terminal when you use the application and there is no panic.

Clean package APIs

A quick glance at the API (in this context, the exposed variables, methods, and types) for our `trace` package highlights that a simple and obvious design has emerged:

- The `New()` – method-creates a new instance of a Tracer
- The `Off()` – method-gets a Tracer that does nothing
- The `Tracer` interface – describes the methods Tracer objects will implement

I would be very confident to give this package to a Go programmer without any documentation or guidelines, and I'm pretty sure they would know what do to with it.



In Go, adding documentation is as simple as adding comments to the line before each item. The blog post on the subject is a worthwhile read (<http://blog.golang.org/godoc-documenting-go-code>), where you can see a copy of the hosted source code for `tracer.go` that is an example of how you might annotate the `trace` package. For more information, refer to <https://github.com/matryer/goblueprints/blob/master/chapter1/tracer/tracer.go>.

Summary

In this chapter, we developed a complete concurrent chat application and our own simple package to trace the flow of our programs to help us better understand what is going on under the hood.

We used the `net/http` package to quickly build what turned out to be a very powerful concurrent HTTP web server. In one particular case, we then upgraded the connection to open a web socket between the client and server. This means that we can easily and quickly communicate messages to the user's web browser without having to write messy polling code. We explored how templates are useful to separate the code from the content as well as to allow us to inject data into our template source, which let us make the host address configurable. Command-line flags helped us give simple configuration control to the people hosting our application while also letting us specify sensible defaults.

Our chat application made use of Go's powerful concurrency capabilities that allowed us to write clear *threaded* code in just a few lines of idiomatic Go. By controlling the coming and going of clients through channels, we were able to set synchronization points in our code that prevented us from corrupting memory by attempting to modify the same objects at the same time.

We learned how interfaces such as `http.Handler` and our own `trace.Tracer` interface allow us to provide disparate implementations without having to touch the code that makes use of them, and in some cases, without having to expose even the name of the implementation to our users. We saw how just by adding a `ServeHTTP` method to our `room` type, we turned our custom room concept into a valid HTTP handler object, which managed our web socket connections.

We aren't actually very far away from being able to properly release our application, except for one major oversight: you cannot see who sent each message. We have no concept of users or even usernames, and for a real chat application, this is not acceptable.

In the next chapter, we will add the names of the people responding to their messages in order to make them feel like they are having a real conversation with other humans.

2

Adding User Accounts

The chat application we built in the previous chapter focused on high performance transmission of messages from the clients to the server and back again. However, the way things stand, our users have no way of knowing who they would be talking to. One solution to this problem is building some kind of sign-up and login functionality and letting our users create accounts and authenticate themselves before they can open the chat page.

Whenever we are about to build something from scratch, we must ask ourselves how others have solved this problem before (it is extremely rare to encounter genuinely original problems) and whether any open solutions or standards already exist that we can make use of. Authorization and authentication can hardly be considered new problems, especially in the world of the Web, with many different protocols out there to choose from. So how do we decide the best option to pursue? As always, we must look at this question from the point of view of the user.

A lot of websites these days allow you to sign in using your accounts that exist elsewhere on a variety of social media or community websites. This saves users the tedious job of entering all of their account information over and over again as they decide to try out different products and services. It also has a positive effect on the conversion rates for new sites.

In this chapter, we will enhance our chat codebase to add authorization, which will allow our users to sign in using Google, Facebook, or GitHub, and you'll see how easy it is to add other sign-in portals too. In order to join the chat, users must first sign in. Following this, we will use the authorized data to augment our user experience so everyone knows who is in the room and who said what.

In this chapter, you will learn to:

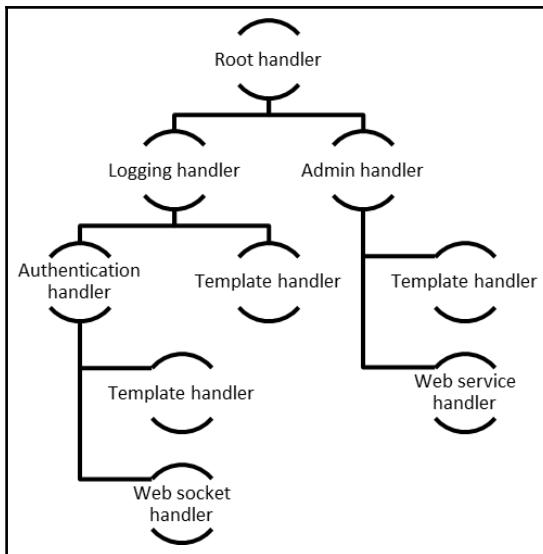
- Use the decorator pattern to wrap `http.Handler` types in order to add additional functionality to handlers
- Serve HTTP endpoints with dynamic paths
- Use the `gomniauth` open source project to access authentication services
- Get and set cookies using the `http` package
- Encode objects as Base64 and back to normal again
- Send and receive JSON data over a web socket
- Give different types of data to templates
- Work with the channels of your own types

Handlers all the way down

For our chat application, we implemented our own `http.Handler` type (the room) in order to easily compile, execute, and deliver HTML content to browsers. Since this is a very simple but powerful interface, we are going to continue to use it wherever possible when adding functionality to our HTTP processing.

In order to determine whether a user is allowed to proceed, we will create an authorization wrapper handler that will perform the check and pass the execution on to the inner handler only if the user is authorized.

Our wrapper handler will satisfy the same `http.Handler` interface as the object inside it, allowing us to wrap any valid handler. In fact, even the authentication handler we are about to write could be later encapsulated inside a similar wrapper if required.



Chaining pattern when applied to HTTP handlers

The preceding diagram shows how this pattern could be applied in a more complicated HTTP handler scenario. Each object implements the `http.Handler` interface. This means that an object could be passed to the `http.Handle` method to directly handle a request, or it can be given to another object, which could add some kind of extra functionality. The `Logging` handler may write to a log file before and after the `ServeHTTP` method is called on the inner handler. Because the inner handler is just another `http.Handler`, any other handler can be wrapped in (or decorated with) the `Logging` handler.

It is also common for an object to contain logic that decides which inner handler should be executed. For example, our authentication handler will either pass the execution to the wrapped handler, or handle the request itself by issuing a redirect to the browser.

That's plenty of theory for now; let's write some code. Create a new file called `auth.go` in the `chat` folder:

```
package main
import ("net/http")
type authHandler struct {
    next http.Handler
}
func (h *authHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    _, err := r.Cookie("auth")
    if err == http.ErrNoCookie {
        // not authenticated
    }
}
```

```
w.Header().Set("Location", "/login")
w.WriteHeader(http.StatusTemporaryRedirect)
return
}
if err != nil {
    // some other error
    http.Error(w, err.Error(), http.StatusInternalServerError)
    return
}
// success - call the next handler
h.next.ServeHTTP(w, r)
}
func MustAuth(handler http.Handler) http.Handler {
    return &authHandler{next: handler}
}
```

The authHandler type not only implements the ServeHTTP method (which satisfies the http.Handler interface), but also stores (wraps) http.Handler in the next field. Our MustAuth helper function simply creates authHandler that wraps any other http.Handler. This is the pattern that allows us to easily add authorization to our code in main.go.

Let's tweak the following root mapping line:

```
http.Handle("/", &templateHandler{filename: "chat.html"})
```

Let's change the first argument to make it explicit about the page meant for chatting. Next, let's use the MustAuth function to wrap templateHandler for the second argument:

```
http.Handle("/chat", MustAuth(&templateHandler{filename: "chat.html"}))
```

Wrapping templateHandler with the MustAuth function will cause the execution to run through authHandler first; it will run only to templateHandler if the request is authenticated.

The ServeHTTP method in authHandler will look for a special cookie called auth, and it will use the Header and WriteHeader methods on http.ResponseWriter to redirect the user to a login page if the cookie is missing. Notice that we discard the cookie itself using the underscore character and capture only the returning error; this is because we only care about whether the cookie is present at this point.

Build and run the chat application and try to hit `http://localhost:8080/chat`:

```
go build -o chat
./chat -host=:8080"
```



You need to delete your cookies to clear out previous authentication tokens or any other cookies that might be left over from other development projects served through the localhost.

If you look in the address bar of your browser, you will notice that you are immediately redirected to the `/login` page. Since we cannot handle that path yet, you'll just get a **404 page not found** error.

Making a pretty social sign-in page

So far, we haven't paid much attention to making our application look nice; after all, this book is about Go and not user interface development. However, there is no excuse for building ugly apps, and so we will build a social sign-in page that is as pretty as it is functional.

Bootstrap is a frontend framework for developing responsive projects on the Web. It provides CSS and JavaScript code that solve many user interface problems in a consistent and good-looking way. While sites built using Bootstrap tend to look the same (although there are a plenty of ways in which the UI can be customized), it is a great choice for early versions of apps or for developers who don't have access to designers.



If you build your application using the semantic standards set forth by Bootstrap, it will become easy for you to make a Bootstrap theme for your site or application, and you know it will slot right into your code.

We will use the version of Bootstrap hosted on a CDN so we don't have to worry about downloading and serving our own version through our chat application. This means that in order to render our pages properly, we will need an active Internet connection even during development.

If you prefer to download and host your own copy of Bootstrap, you can do so. Keep the files in an `assets` folder and add the following call to your `main` function (it uses `http.Handle` to serve the assets via your application):

```
http.Handle("/assets/", http.StripPrefix("/assets",
    http.FileServer(http.Dir("/path/to/assets/"))))
```



Notice how the `http.StripPrefix` and `http.FileServer` functions return objects that satisfy the `http.Handler` interface as per the decorator pattern that we implement with our `MustAuth` helper function.

In `main.go`, let's add an endpoint for the login page:

```
http.Handle("/chat", MustAuth(&templateHandler{filename: "chat.html"}))
http.Handle("/login", &templateHandler{filename: "login.html"})
http.Handle("/room", r)
```

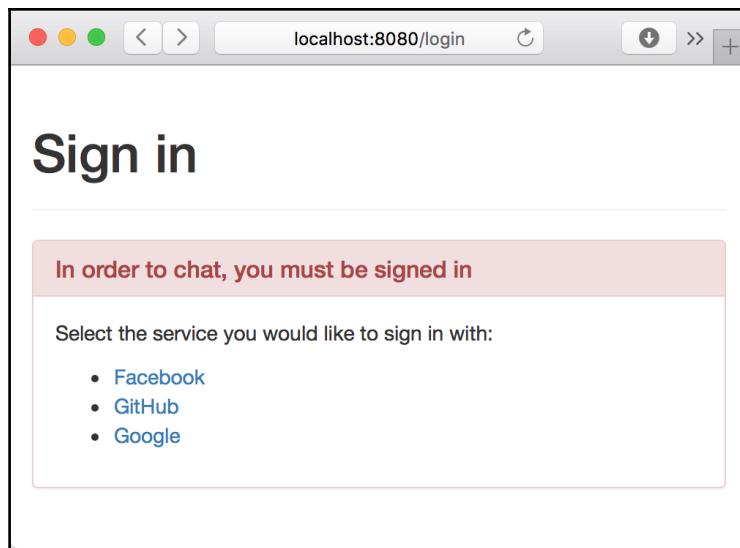
Obviously, we do not want to use the `MustAuth` method for our login page because it will cause an infinite redirection loop.

Create a new file called `login.html` inside our `templates` folder and insert the following HTML code:

```
<html>
  <head>
    <title>Login</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css">
  </head>
  <body>
    <div class="container">
      <div class="page-header">
        <h1>Sign in</h1>
      </div>
      <div class="panel panel-danger">
        <div class="panel-heading">
          <h3 class="panel-title">In order to chat, you must be signed in</h3>
        </div>
        <div class="panel-body">
          <p>Select the service you would like to sign in with:</p>
          <ul>
            <li>
              <a href="/auth/login/facebook">Facebook</a>
            </li>
            <li>
              <a href="/auth/login/github">GitHub</a>
            </li>
            <li>
              <a href="/auth/login/google">Google</a>
            </li>
          </ul>
        </div>
      </div>
    </div>
  </body>
</html>
```

```
</div>
</div>
</body>
</html>
```

Restart the web server and navigate to `http://localhost:8080/login`. You will notice that it now displays our **Sign in** page:



Endpoints with dynamic paths

Pattern matching for the `http` package in the Go standard library isn't the most comprehensive and fully featured implementation out there. For example, Ruby on Rails makes it much easier to have dynamic segments inside the path. You could map the route like this:

```
"auth/:action/:provider_name"
```

Rails then provides a data map (or dictionary) containing the values that it automatically extracted from the matched path. So if you visit `auth/login/google`, then `params[:provider_name]` would equal `google` and `params[:action]` would equal `login`.

The most the `http` package lets us specify by default is a path prefix, which we can make use of by leaving a trailing slash at the end of the pattern:

```
"auth/"
```

We would then have to manually parse the remaining segments to extract the appropriate data. This is acceptable for relatively simple cases. This suits our needs for the time being since we only need to handle a few different paths, such as the following:

- `/auth/login/google`
- `/auth/login/facebook`
- `/auth/callback/google`
- `/auth/callback/facebook`



If you need to handle more advanced routing situations, you may want to consider using dedicated packages, such as `goweb`, `pat`, `routes`, or `mux`. For extremely simple cases such as ours, built-in capabilities will do.

We are going to create a new handler that powers our login process. In `auth.go`, add the following `loginHandler` code:

```
// loginHandler handles the third-party login process.
// format: /auth/{action}/{provider}
func loginHandler(w http.ResponseWriter, r *http.Request) {
    segs := strings.Split(r.URL.Path, "/")
    action := segs[2]
    provider := segs[3]
    switch action {
    case "login":
        log.Println("TODO handle login for", provider)
    default:
        w.WriteHeader(http.StatusNotFound)
        fmt.Fprintf(w, "Auth action %s not supported", action)
    }
}
```

In the preceding code, we break the path into segments using `strings.Split` before pulling out the values for `action` and `provider`. If the `action` value is known, we will run the specific code; otherwise, we will write out an error message and return an `http.StatusNotFound` status code (which in the language of HTTP status code is 404).



We will not bulletproof our code right now. But it's worth noticing that if someone hits `loginHandler` with few segments, our code will panic because it would expect `segs[2]` and `segs[3]` to exist. For extra credit, see whether you can protect your code against this and return a nice error message instead of making it panic if someone hits `/auth/nonsense`.

Our `loginHandler` is only a function and not an object that implements the `http.Handler` interface. This is because, unlike other handlers, we don't need it to store any state. The Go standard library supports this, so we can use the `http.HandleFunc` function to map it in a way similar to how we used `http.Handle` earlier. In `main.go`, update the handlers:

```
http.Handle("/chat", MustAuth(&templateHandler{filename: "chat.html"}))
http.Handle("/login", &templateHandler{filename: "login.html"})
http.HandleFunc("/auth/", loginHandler)
http.Handle("/room", r)
```

Rebuild and run the chat application:

```
go build -o chat
./chat -host=:8080"
```

Hit the following URLs and notice the output logged in the terminal:

- `http://localhost:8080/auth/login/google` outputs TODO handle login for google
- `http://localhost:8080/auth/login/facebook` outputs TODO handle login for facebook

We have successfully implemented a dynamic path-matching mechanism that just prints out TODO messages so far; we need to integrate it with authorization services in order to make our login process work.

Getting started with OAuth2

OAuth2 is an open authorization standard designed to allow resource owners to give clients delegated access to private data (such as wall posts or tweets) via an access token exchange handshake. Even if you do not wish to access the private data, OAuth2 is a great option that allows people to sign in using their existing credentials, without exposing those credentials to a third-party site. In this case, we are the third party, and we want to allow our users to sign in using services that support OAuth2.

From a user's point of view, the OAuth2 flow is as follows:

1. The user selects the provider with whom they wish to sign in to the client app.
2. The user is redirected to the provider's website (with a URL that includes the client app ID) where they are asked to give permission to the client app.
3. The user signs in from the OAuth2 service provider and accepts the permissions requested by the third-party application.
4. The user is redirected to the client app with a request code.
5. In the background, the client app sends the grant code to the provider, who sends back an authentication token.
6. The client app uses the access token to make authorized requests to the provider, such as to get user information or wall posts.

To avoid reinventing the wheel, we will look at a few open source projects that have already solved this problem for us.

Open source OAuth2 packages

Andrew Gerrand has been working on the core Go team since February 2010, that is, two years before Go 1.0 was officially released. His `golang/oauth2` package (see <https://github.com/golang/oauth2>) is an elegant implementation of the OAuth2 protocol written entirely in Go.

Andrew's project inspired `gomniauth` (see <https://github.com/stretchr/gomniauth>). An open source Go alternative to Ruby's `omniauth` project, `gomniauth` provides a unified solution to access different OAuth2 services. In the future, when OAuth3 (or whatever the next-generation authorization protocol will be) comes out, in theory `gomniauth` could take on the pain of implementing the details, leaving the user code untouched.

For our application, we will use `gomniauth` to access OAuth services provided by Google, Facebook, and GitHub, so make sure you have it installed by running the following command:

```
go get github.com/stretchr/gomniauth
```



Some of the project dependencies of `gomniauth` are kept in Bazaar repositories, so you'll need to head over to <http://wiki.bazaar.canonical.com> to download them.

Tell the authorization providers about your app

Before we ask an authorization provider to help our users sign in, we must tell them about our application. Most providers have some kind of web tool or console where you can create applications to kick this process off. Here's one from Google:

The screenshot shows a 'New Project' dialog box. It has fields for 'Project name' (containing 'Chat Application'), 'Project ID' (containing 'blueprints-chat-app'), and 'Show advanced options...'. At the bottom are 'Create' and 'Cancel' buttons. The 'Project ID' field is highlighted with a red border.

In order to identify the client application, we need to create a client ID and secret. Despite the fact that OAuth2 is an open standard, each provider has their own language and mechanism to set things up. Therefore, you will most likely have to play around with the user interface or the documentation to figure it out in each case.

At the time of writing, in **Google Cloud Console**, you navigate to **API Manager** and click on the **Credentials** section.

In most cases, for added security, you have to be explicit about the host URLs from where requests will come. For now, since we're hosting our app locally on `localhost:8080`, you should use it. You will also be asked for a redirect URI that is the endpoint in our chat application and to which the user will be redirected after they successfully sign in. The callback will be another action in `loginHandler`, so the redirect URL for the Google client will be `http://localhost:8080/auth/callback/google`.

Once you finish the authorization process for the providers you want to support, you will be given a client ID and secret for each provider. Make a note of these details because we will need them when we set up the providers in our chat application.



If we host our application on a real domain, we have to create new client IDs and secrets or update the appropriate URL fields on our authorization providers to ensure that they point to the right place. Either way, it is good practice to have a different set of development and production keys for security.

Implementing external logging in

In order to make use of the projects, clients, or accounts that we created on the authorization provider sites, we have to tell `gomniauth` which providers we want to use and how we will interact with them. We do this by calling the `WithProviders` function on the primary `gomniauth` package. Add the following code snippet to `main.go` (just underneath the `flag.Parse()` line toward the top of the `main` function):

```
// setup gomniauth
gomniauth.SetSecurityKey("PUT YOUR AUTH KEY HERE")
gomniauth.WithProviders(
    facebook.New("key", "secret",
        "http://localhost:8080/auth/callback/facebook"),
    github.New("key", "secret",
        "http://localhost:8080/auth/callback/github"),
    google.New("key", "secret",
        "http://localhost:8080/auth/callback/google"),
)
```

You should replace the `key` and `secret` placeholders with the actual values you noted down earlier. The third argument represents the callback URL that should match the ones you provided when creating your clients on the provider's website. Notice the second path segment is `callback`; while we haven't implemented this yet, this is where we handle the response from the authorization process.

As usual, you will need to ensure all the appropriate packages are imported:

```
import (
    "github.com/stretchr/gomniauth/providers/facebook"
    "github.com/stretchr/gomniauth/providers/github"
    "github.com/stretchr/gomniauth/providers/google"
)
```



Gomniauth requires the `SetSecurityKey` call because it sends state data between the client and server along with a signature checksum, which ensures that the state values are not tempered with while being transmitted. The security key is used when creating the hash in a way that it is almost impossible to recreate the same hash without knowing the exact security key. You should replace `some long key` with a security hash or phrase of your choice.

Logging in

Now that we have configured Gomniauth, we need to redirect users to the provider's authorization page when they land on our `/auth/login/{provider}` path. We just have to update our `loginHandler` function in `auth.go`:

```
func loginHandler(w http.ResponseWriter, r *http.Request) {
    segs := strings.Split(r.URL.Path, "/")
    action := segs[2]
    provider := segs[3]
    switch action {
    case "login":
        provider, err := gomniauth.Provider(provider)
        if err != nil {
            http.Error(w, fmt.Sprintf("Error when trying to get provider
                %s: %s", provider, err), http.StatusBadRequest)
            return
        }
        loginUrl, err := provider.GetBeginAuthURL(nil, nil)
        if err != nil {
            http.Error(w, fmt.Sprintf("Error when trying to GetBeginAuthURL
                for %s: %s", provider, err), http.StatusInternalServerError)
        }
    }
}
```

```
        return
    }
    w.Header.Set("Location", loginUrl)
    w.WriteHeader(http.StatusTemporaryRedirect)
    default:
        w.WriteHeader(http.StatusNotFound)
        fmt.Fprintf(w, "Auth action %s not supported", action)
    }
}
```

We do two main things here. First, we use the `gomniauth.Provider` function to get the provider object that matches the object specified in the URL (such as `google` or `github`). Then, we use the `GetBeginAuthURL` method to get the location where we must send users to in order to start the authorization process.



The `GetBeginAuthURL(nil, nil)` arguments are for the state and options respectively, which we are not going to use for our chat application. The first argument is a state map of data that is encoded and signed and sent to the authentication provider. The provider doesn't do anything with the state; it just sends it back to our callback endpoint. This is useful if, for example, we want to redirect the user back to the original page they were trying to access before the authentication process intervened. For our purpose, we have only the `/chat` endpoint, so we don't need to worry about sending any state. The second argument is a map of additional options that will be sent to the authentication provider, which somehow modifies the behavior of the authentication process. For example, you can specify your own `scope` parameter, which allows you to make a request for permission to access additional information from the provider. For more information about the available options, search for OAuth2 on the Internet or read the documentation for each provider, as these values differ from service to service.

If our code gets no error from the `GetBeginAuthURL` call, we simply redirect the user's browser to the returned URL.

If errors occur, we use the `http.Error` function to write the error message out with a non-200 status code.

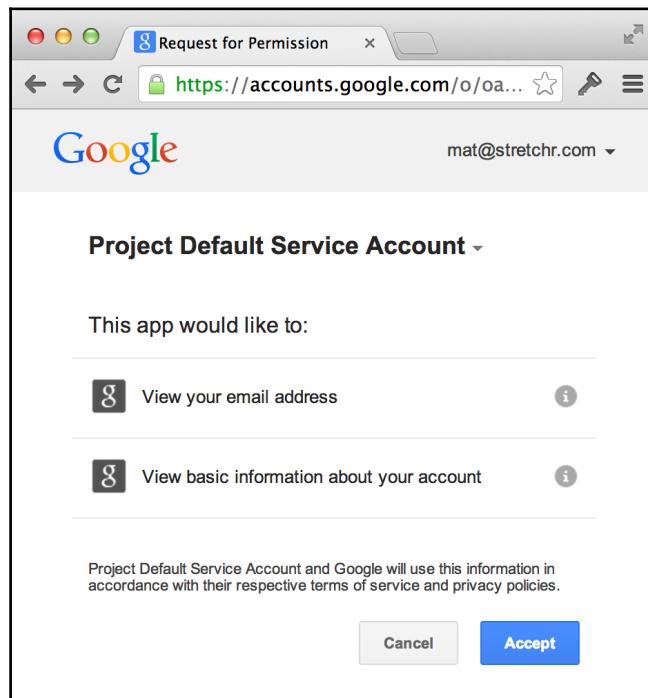
Rebuild and run the chat application:

```
go build -o chat  
./chat -host=:8080
```



We will continue to stop, rebuild, and run our projects manually throughout this book, but there are some tools that will take care of this for you by watching for changes and restarting Go applications automatically. If you're interested in such tools, check out <https://github.com/pilu/freshand> <https://github.com/codegangsta/gin>.

Open the main chat page by accessing <http://localhost:8080/chat>. As we aren't logged in yet, we are redirected to our sign-in page. Click on the **Google** option to sign in using your Google account and you will notice that you are presented with a Google-specific sign-in page (if you are not already signed in to Google). Once you are signed in, you will be presented with a page asking you to give permission for our chat application before you can view basic information about your account:



This is the same flow that the users of our chat application will experience when signing in.

Click on **Accept** and you will notice that you are redirected to our application code but presented with an `Auth` action callback not supported error. This is because we haven't yet implemented the callback functionality in `loginHandler`.

Handling the response from the provider

Once the user clicks on **Accept** on the provider's website (or if they click on the equivalent of **Cancel**), they will be redirected to the callback endpoint in our application.

A quick glance at the complete URL that comes back shows us the grant code that the provider has given us:

```
http://localhost:8080/auth/callback/google?code=4/Q92xJ-BQfoX6PHhzkjhgtyfLc0Ylm.QqV4u9AbA9sYguyfbjFEsNoJKMOjQI
```

We don't have to worry about what to do with this code because Gomniauth does it for us; we can simply jump to implementing our callback handler. However, it's worth knowing that this code will be exchanged by the authentication provider for a token that allows us to access private user data. For added security, this additional step happens behind the scenes, from server to server rather than in the browser.

In `auth.go`, we are ready to add another switch case to our action path segment. Insert the following code before the default case:

```
case "callback":
    provider, err := gomniauth.Provider(provider)
    if err != nil {
        http.Error(w, fmt.Sprintf("Error when trying to get provider %s: %s",
            provider, err), http.StatusBadRequest)
        return
    }
    creds, err :=
    provider.CompleteAuth(objx.MustFromURLQuery(r.URL.RawQuery))
    if err != nil {
        http.Error(w, fmt.Sprintf("Error when trying to complete auth for
            %s: %s", provider, err), http.StatusInternalServerError)
        return
    }
    user, err := provider.GetUser(creds)
    if err != nil {
        http.Error(w, fmt.Sprintf("Error when trying to get user from %s: %s",
            provider, err), http.StatusInternalServerError)
        return
    }
    authCookieValue := objx.New(map[string]interface{}{
```

```
    "name": user.Name(),
}) .MustBase64()
http.SetCookie(w, &http.Cookie{
    Name:  "auth",
    Value: authCookieValue,
    Path:   "/"})
w.Header().Set("Location", "/chat")
w.WriteHeader(http.StatusTemporaryRedirect)
```

When the authentication provider redirects the users after they have granted permission, the URL specifies that it is a callback action. We look up the authentication provider as we did before and call its `CompleteAuth` method. We parse `RawQuery` from the request into `objx.Map` (the multipurpose map type that `Gomniauth` uses), and the `CompleteAuth` method uses the values to complete the OAuth2 provider handshake with the provider. All being well, we will be given some authorized credentials with which we will be able to access our user's basic data. We then use the `GetUser` method for the provider, and `Gomniauth` will use the specified credentials to access some basic information about the user.

Once we have the user data, we **Base64-encode** the `Name` field in a JSON object and store it as a value for our `auth` cookie for later use.



Base64-encoding data ensures it won't contain any special or unpredictable characters, which is useful for situations such as passing data to a URL or storing it in a cookie. Remember that although Base64-encoded data looks encrypted, it is not you can easily decode Base64-encoded data back to the original text with little effort. There are online tools that do this for you.

After setting the cookie, we redirect the user to the chat page, which we can safely assume was the original destination.

Once you build and run the code again and hit the `/chat` page, you will notice that the sign up flow works and we are finally allowed back to the chat page. Most browsers have an inspector or a console—a tool that allows you to view the cookies that the server has sent you—that you can use to see whether the `auth` cookie has appeared:

```
go build -o chat
./chat -host=:8080"
```

In our case, the cookie value is `eyJyYW1IjoiTWF0IFJ5ZXIifQ==`, which is a Base64-encoded version of `{"name": "Mat Ryer"}`. Remember, we never typed in a name in our chat application; instead, Gomniauth asked Google for a name when we opted to sign in with Google. Storing non-signed cookies like this is fine for incidental information, such as a user's name; however, you should avoid storing any sensitive information using non-signed cookies as it's easy for people to access and change the data.

Presenting the user data

Having the user data inside a cookie is a good start, but non-technical people will never even know it's there, so we must bring the data to the fore. We will do this by enhancing `templateHandler` that first passes the user data to the template's `Execute` method; this allows us to use template annotations in our HTML to display the user data to the users.

Update the `ServeHTTP` method of `templateHandler` in `main.go`:

```
func (t *templateHandler) ServeHTTP(w http.ResponseWriter, r
    *http.Request) {
    t.once.Do(func() {
        t.templ = template.Must(template.ParseFiles(filepath.Join("templates",
            t.filename)))
    })
    data := map[string]interface{}{
        "Host": r.Host,
    }
    if authCookie, err := r.Cookie("auth"); err == nil {
        data["UserData"] = objx.MustFromBase64(authCookie.Value)
    }
    t.templ.Execute(w, data)
}
```

Instead of just passing the entire `http.Request` object to our template as `data`, we are creating a new `map[string]interface{}` definition for a `data` object that potentially has two fields: `Host` and `UserData` (the latter will only appear if an `auth` cookie is present). By specifying the `map` type followed by curly braces, we are able to add the `Host` entry at the same time as making our map while avoiding the `make` keyword altogether. We then pass this new `data` object as the second argument to the `Execute` method on our template.

Now we add an HTML file to our template source to display the name. Update the chatbox form in `chat.html`:

```
<form id="chatbox">
  {{.UserData.name}}:<br/>
  <textarea></textarea>
  <input type="submit" value="Send" />
</form>
```

The `{{.UserData.name}}` annotation tells the template engine to insert our user's name before the `textarea` control.

Since we're using the `objx` package, don't forget to run `go get http://github.com/stretchr/objx` and import it. Additional dependencies add complexity to projects, so you may decide to copy and paste the appropriate functions from the package or even write your own code that marshals between Base64-encoded cookies and back.

Alternatively, you can **vendor** the dependency by copying the whole source code to your project (inside a root-level folder called `vendor`). Go will, at build time, first check the `vendor` folder for any imported packages before checking them in `$GOPATH` (which were put there by `go get`). This allows you to fix the exact version of a dependency rather than rely on the fact that the source package hasn't changed since you wrote your code. For more information about using vendors in Go, check out Daniel Theophanes' post on the subject at <https://blog.gopheracademy.com/advent-2015/vendor-folder/> or search for `vendoring` in Go.



Rebuild and run the chat application again and you will notice the addition of your name before the chat box:

```
go build -o chat
./chat -host=:8080
```

Augmenting messages with additional data

So far, our chat application has only transmitted messages as slices of bytes or `[]byte` types between the client and the server; therefore, the forward channel for our room has the `chan []byte` type. In order to send data (such as who sent it and when) in addition to the message itself, we enhance our forward channel and also how we interact with the web socket on both ends.

Define a new type that will replace the `[]byte` slice by creating a new file called `message.go` in the `chat` folder:

```
package main
import (
    "time"
)
// message represents a single message
type message struct {
    Name    string
    Message string
    When    time.Time
}
```

The `message` type will encapsulate the message string itself, but we have also added the `Name` and `When` fields that respectively hold the user's name and a timestamp of when the message was sent.

Since the `client` type is responsible for communicating with the browser, it needs to transmit and receive more than just a single message string. As we are talking to a JavaScript application (that is, the chat client running in the browser) and the Go standard library has a great JSON implementation, this seems like the perfect choice to encode additional information in the messages. We will change the `read` and `write` methods in `client.go` to use the `ReadJSON` and `WriteJSON` methods on the socket, and we will encode and decode our new message type:

```
func (c *client) read() {
    defer c.socket.Close()
    for {
        var msg *message
        err := c.socket.ReadJSON(&msg)
        if err != nil {
            return
        }
        msg.When = time.Now()
        msg.Name = c.userData["name"].(string)
        c.room.forward <- msg
    }
}
func (c *client) write() {
    defer c.socket.Close()
    for msg := range c.send {
        err := c.socket.WriteJSON(msg)
        if err != nil {
            break
        }
}
```

```
}
```

When we receive a message from the browser, we will expect to populate only the `Message` field, which is why we set the `When` and `Name` fields ourselves in the preceding code.

You will notice that when you try to build the preceding code, it complains about a few things. The main reason is that we are trying to send a `*message` object down our `forward` and `send chan []byte` channels. This is not allowed until we change the type of the channel. In `room.go`, change the `forward` field to be of the type `chan *message`, and do the same for the `send chan` type in `client.go`.

We must update the code that initializes our channels since the types have now changed. Alternatively, you can wait for the compiler to raise these issues and fix them as you go. In `room.go`, you need to make the following changes:

- Change `forward: make(chan []byte)` to `forward: make(chan *message)`
- Change `r.tracer.Trace("Message received: ", string(msg))` to `r.tracer.Trace("Message received: ", msg.Message)`
- Change `send: make(chan []byte, messageBufferSize)` to `send: make(chan *message, messageBufferSize)`

The compiler will also complain about the lack of user data on the client, which is a fair point because the `client` type has no idea about the new user data we have added to the cookie. Update the `client` struct to include a new general-purpose

`map[string]interface{}` called `userData`:

```
// client represents a single chatting user.
type client struct {
    // socket is the web socket for this client.
    socket *websocket.Conn
    // send is a channel on which messages are sent.
    send chan *message
    // room is the room this client is chatting in.
    room *room
    // userData holds information about the user
    userData map[string]interface{}
}
```

The user data comes from the client cookie that we access through the `http.Request` object's `Cookie` method. In `room.go`, update `ServeHTTP` with the following changes:

```
func (r *room) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    socket, err := upgrader.Upgrade(w, req, nil)
    if err != nil {
        log.Fatal("ServeHTTP:", err)
        return
    }
    authCookie, err := req.Cookie("auth")
    if err != nil {
        log.Fatal("Failed to get auth cookie:", err)
        return
    }
    client := &client{
        socket:    socket,
        send:      make(chan *message, messageBufferSize),
        room:      r,
        userData: objx.MustFromBase64(authCookie.Value),
    }
    r.join <- client
    defer func() { r.leave <- client }()
    go client.write()
    client.read()
}
```

We use the `Cookie` method on the `http.Request` type to get our user data before passing it to the client. We are using the `objx.MustFromBase64` method to convert our encoded cookie value back into a usable map object.

Now that we have changed the type being sent and received on the socket from `[]byte` to `*message`, we must tell our JavaScript client that we are sending JSON instead of just a plain string. Also, we must ask that it send JSON back to the server when a user submits a message. In `chat.html`, first update the `socket.send` call:

```
socket.send(JSON.stringify({ "Message": msgBox.val() }));
```

We are using `JSON.stringify` to serialize the specified JSON object (containing just the `Message` field) into a string, which is then sent to the server. Our Go code will decode (or unmarshal) the JSON string into a `message` object, matching the field names from the client JSON object with those of our `message` type.

Finally, update the `socket.onmessage` callback function to expect JSON, and also add the name of the sender to the page:

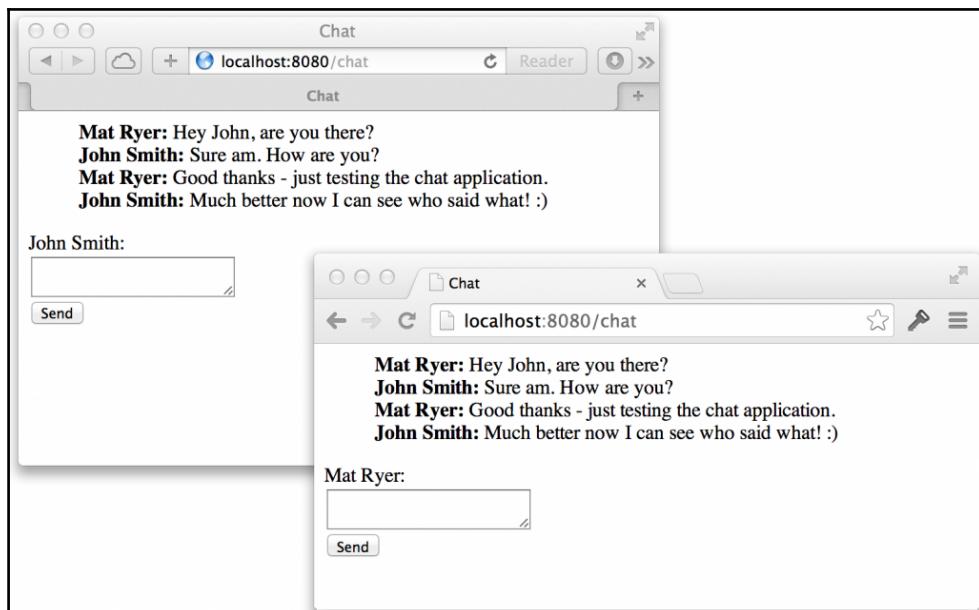
```
socket.onmessage = function(e) {
  var msg = JSON.parse(e.data);
  messages.append(
    $("<li>").append(
      $("<strong>").text(msg.Name + ": "),
      $("<span>").text(msg.Message)
    )
  );
}
```

In the preceding code snippet, we used JavaScript's `JSON.parse` function to turn the JSON string into a JavaScript object and then access the fields to build up the elements needed to properly display them.

Build and run the application, and if you can, log in with two different accounts in two different browsers (or invite a friend to help you test it):

```
go build -o chat
./chat -host=:8080"
```

The following screenshot shows the chat application's browser chat screens:



Summary

In this chapter, we added a useful and necessary feature to our chat application by asking users to authenticate themselves using OAuth2 service providers before we allow them to join the conversation. We made use of several open source packages, such as `Gomniauth`, which dramatically reduced the amount of multiserver complexity we would otherwise have dealt with.

We implemented a pattern when we wrapped `http.Handler` types to allow us to easily specify which paths require the user to be authenticated and which were available, even without an `auth` cookie. Our `MustAuth` helper function allowed us to generate the wrapper types in a fluent and simple way, without adding clutter and confusion to our code.

We saw how to use cookies and Base64-encoding to safely (although not securely) store the state of particular users in their respective browsers and to make use of that data over normal connections and through web sockets. We took more control of the data available to our templates in order to provide the name of the user to the UI and saw how to only provide certain data under specific conditions.

Since we needed to send and receive additional information over the web socket, we learned how easy it was to change the channels of native types into channels that work with types of our own, such as our `message` type. We also learned how to transmit JSON objects over the socket, rather than just slices of bytes. Thanks to the type safety of Go and the ability to specify types for channels, the compiler helps ensure that we do not send anything other than `message` objects through `chan *message`. Attempting to do so would result in a compiler error, alerting us to the fact right away.

From building a chat application to seeing the name of the person chatting is a great leap forward in terms of usability. But it's very formal and might not attract modern users of the Web, who are used to a much more visual experience. We are missing pictures of people chatting, and in the next chapter, we will explore different ways in which this could be done. We can allow users to better represent themselves in our application by pulling profile pictures (avatars) from the OAuth2 provider, the Gravatar web service, or the local disk after the users have uploaded them.

As an extra assignment, see whether you can make use of the `time.Time` field that we put into the `message` type to tell users when the messages were sent.

3

Three Ways to Implement Profile Pictures

So far, our chat application has made use of the **OAuth2** protocol to allow users to sign in to our application so that we know who is saying what. In this chapter, we are going to add profile pictures to make the chatting experience more engaging.

We will look at the following ways to add pictures or avatars alongside the messages in our application:

- Using the avatar picture provided by the auth service
- Using the <https://en.gravatar.com/> web service to look up a picture by the user's e-mail address
- Allowing the user to upload their own picture and host it themselves

The first two options allow us to delegate the hosting of pictures to a third party either an authorization service or <https://en.gravatar.com/> which is great because it reduces the cost of hosting our application (in terms of storage costs and bandwidth, since the user's browsers will actually download the pictures from the servers of the authenticating service, not ours). The third option requires us to host pictures ourselves at a location that is accessible on the Web.

These options aren't mutually exclusive; you will most likely use a combination of them in a real-world production application. Toward the end of the chapter, you will see how the flexible design that emerges allows us to try each implementation in turn until we find an appropriate avatar.

We are going to be agile with our design throughout this chapter, doing the minimum work needed to accomplish each milestone. This means that at the end of each section, we will have working implementations that are demonstrable in the browser. This also means that we will refactor code as and when we need to and discuss the rationale behind the decisions we make as we go.

Specifically, in this chapter, you will learn:

- What the good practices to get additional information from auth services are, even when there are no standards in place
- When it is appropriate to build abstractions into our code
- How Go's zero-initialization pattern can save time and memory
- How reusing an interface allows us to work with collections and individual objects in the same way as the existing interface did
- How to use the <https://en.gravatar.com/> web service
- How to do MD5 hashing in Go
- How to upload files over HTTP and store them on a server
- How to serve static files through a Go web server
- How to use unit tests to guide the refactoring of code
- How and when to abstract functionality from `struct` types into interfaces

Avatars from the OAuth2 server

It turns out that most auth servers already have images for their users, and they make them available through the protected user resource that we already used in order to get our user's names. To use this avatar picture, we need to get the URL from the provider, store it in the cookie for our user, and send it through a web socket so that every client can render the picture alongside the corresponding message.

Getting the avatar URL

The schema for user or profile resources is not part of the OAuth2 spec, which means that each provider is responsible for deciding how to represent that data. Indeed, providers do things differently; for example, the avatar URL in a GitHub user resource is stored in a field called `avatar_url`, whereas in Google, the same field is called `picture`. Facebook goes even further by nesting the avatar URL value in a `url` field inside an object called `picture`. Luckily, Gomniauth abstracts this for us; its `GetUser` call on a provider standardizes the interface to get common fields.

In order to make use of the avatar URL field, we need to go back and store that information in our cookie. In `auth.go`, look inside the `callback` action switch case and update the code that creates the `authCookieValue` object, as follows:

```
authCookieValue := objx.New(map[string]interface{}{  
    "name": user.Name(),  
    "avatar_url": user.AvatarURL(),  
}).MustBase64()
```

The `AvatarURL` field called in the preceding code will return the appropriate URL value and store it in our `avatar_url` field, which we then put into the cookie.



Gomniauth defines a `User` type of interface and each provider implements their own version. The generic `map[string]interface{}` data returned from the auth server is stored inside each object, and the method calls access the appropriate value using the right field name for that provider. This approach describing the way information is accessed without being strict about implementation details—is a great use of interfaces in Go.

Transmitting the avatar URL

We need to update our message type so that it can also carry the avatar URL with it. In `message.go`, add the `AvatarURL` string field:

```
type message struct {  
    Name      string  
    Message   string  
    When      time.Time  
    AvatarURL string  
}
```

So far, we have not actually assigned a value to `AvatarURL` like we do for the `Name` field; so, we must update our `read` method in `client.go`:

```
func (c *client) read() {  
    defer c.socket.Close()  
    for {  
        var msg *message  
        err := c.socket.ReadJSON(&msg)  
        if err != nil {  
            return  
        }  
        msg.When = time.Now()  
        msg.Name = c.userData["name"].(string)
```

```
        if avatarURL, ok := c.userData["avatar_url"]; ok {
            msg.AvatarURL = avatarURL.(string)
        }
        c.room.forward <- msg
    }
}
```

All we have done here is take the value from the `userData` field that represents what we put into the cookie and assigned it to the appropriate field in `message` if the value was present in the map. We now take the additional step of checking whether the value is present because we cannot guarantee that the auth service would provide a value for this field. And since it could be `nil`, it might cause panic to assign it to a `string` type if it's actually missing.

Adding the avatar to the user interface

Now that our JavaScript client gets an avatar URL value via the socket, we can use it to display the image alongside the messages. We do this by updating the `socket.onmessage` code in `chat.html`:

```
socket.onmessage = function(e) {
    var msg = JSON.parse(e.data);
    messages.append(
        $("<li>").append(
            $("<img>").css({
                width:50,
                verticalAlign:"middle"
            }).attr("src", msg.AvatarURL),
            $("<strong>").text(msg.Name + ": "),
            $("<span>").text(msg.Message)
        )
    );
}
```

When we receive a message, we will insert an `img` tag with the source set to the `AvatarURL` field. We will use jQuery's `css` method to force a width of 50 pixels. This protects us from massive pictures spoiling our interface and allows us to align the image to the middle of the surrounding text.

If we build and run our application having logged in with a previous version, you will find that the auth cookie that doesn't contain the avatar URL is still there. We are not asked to authenticate again (since we are already logged in), and the code that adds the `avatar_url` field never gets a chance to run. We could delete our cookie and refresh the page, but we would have to keep doing this whenever we make changes during development. Let's solve this problem properly by adding a logout feature.

Logging out

The simplest way to log out a user is to get rid of the auth cookie and redirect the user to the chat page, which will in turn cause a redirect to the login page (since we just removed the cookie). We do this by adding a new `HandleFunc` call to `main.go`:

```
http.HandleFunc("/logout", func(w http.ResponseWriter, r *http.Request) {
    http.SetCookie(w, &http.Cookie{
        Name:     "auth",
        Value:    "",
        Path:     "/",
        MaxAge:   -1,
    })
    w.Header().Set("Location", "/chat")
    w.WriteHeader(http.StatusTemporaryRedirect)
})
```

The preceding handler function uses `http.SetCookie` to update the cookie setting `MaxAge` to `-1`, which indicates that it should be deleted immediately by the browser. Not all browsers are forced to delete the cookie, which is why we also provide a new `Value` setting of an empty string, thus removing the user data that would previously have been stored.



As an additional assignment, you can bulletproof your app a little by updating the first line in `ServeHTTP` for your `authHandler` method in `auth.go` to make it cope with the empty value case as well as the missing cookie case: `if cookie, err := r.Cookie("auth"); err == http.ErrNoCookie || cookie.Value == ""` Instead of ignoring the return of `r.Cookie`, we keep a reference to the returned cookie (if there was actually one) and also add an additional check to see whether the `Value` string of the cookie is empty or not.

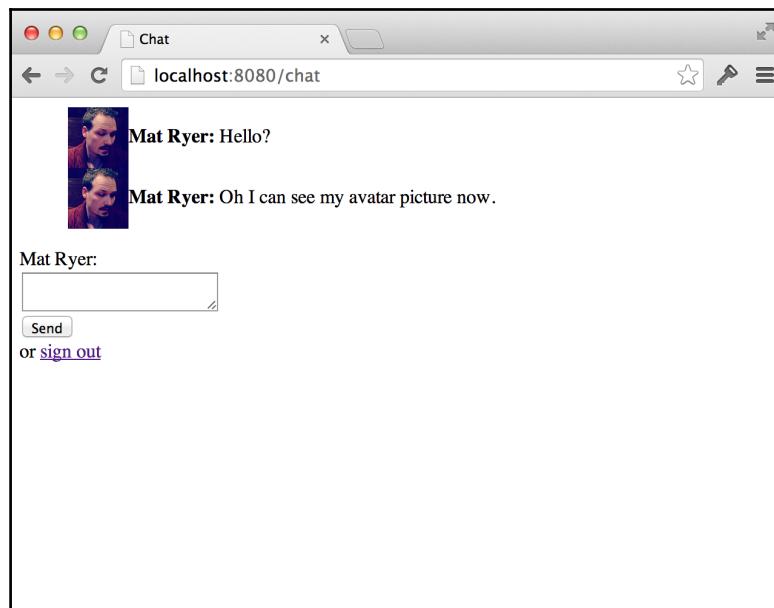
Before we continue, let's add a `Sign Out` link to make it even easier to get rid of the cookie and also allow our end users to log out. In `chat.html`, update the `chatbox` form to insert a simple HTML link to the new `/logout` handler:

```
<form id="chatbox">
  {{UserData.name}}:<br/>
  <textarea></textarea>
  <input type="submit" value="Send" />
  or <a href="/logout">sign out</a>
</form>
```

Now build and run the application and open a browser to `localhost:8080/chat`:

```
go build -o chat
./chat -host=:8080
```

Log out if you need to and log back in. When you click on **Send**, you will see your avatar picture appear next to your messages:



Making things prettier

Our application is starting to look a little ugly, and its time to do something about it. In the previous chapter, we implemented the Bootstrap library into our login page, and we are going to extend its use to our chat page now. We will make three changes in `chat.html`: include Bootstrap and tweak the CSS styles for our page, change the markup for our form, and tweak how we render messages on the page:

1. First, let's update the `style` tag at the top of the page and insert a `link` tag above it in order to include Bootstrap:

```
<link rel="stylesheet" href="//netdna.bootstrapcdncdn.com/bootstrap  
/3.3.6/css/bootstrap.min.css">  
<style>  
    ul#messages { list-style: none; }  
    ul#messages li { margin-bottom: 2px; }  
    ul#messages li img { margin-right: 10px; }  
</style>
```

2. Next, let's replace the markup at the top of the `body` tag (before the `script` tags) with the following code:

```
<div class="container">  
    <div class="panel panel-default">  
        <div class="panel-body">  
            <ul id="messages"></ul>  
        </div>  
    </div>  
    <form id="chatbox" role="form">  
        <div class="form-group">  
            <label for="message">Send a message as {{UserData.name}}  
            </label> or <a href="/logout">Sign out</a>  
            <textarea id="message" class="form-control"></textarea>  
        </div>  
        <input type="submit" value="Send" class="btn btn-default" />  
    </form>  
</div>
```



This markup follows Bootstrap standards of applying appropriate classes to various items; for example, the `form-control` class neatly formats elements within `form` (you can check out the Bootstrap documentation for more information on what these classes do).

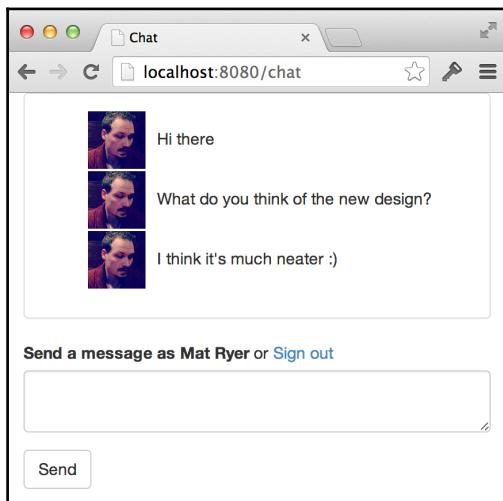
3. Finally, let's update our `socket.onmessage` JavaScript code to put the sender's name as the title attribute for our image. This makes it display the image when you mouse over it rather than display it next to every message:

```
socket.onmessage = function(e) {
  var msg = JSON.parse(e.data);
  messages.append(
    $("<li>").append(
      $("<img>").attr("title", msg.Name).css({
        width:50,
        verticalAlign:"middle"
      }).attr("src", msg.AvatarURL),
      $("<span>").text(msg.Message)
    )
  );
}
```

Build and run the application and refresh your browser to see whether a new design appears:

```
go build -o chat
./chat -host=:8080
```

The preceding command shows the following output:



With relatively few changes to the code, we have dramatically improved the look and feel of our application.

Implementing Gravatar

Gravatar is a web service that allows users to upload a single profile picture and associate it with their e-mail address in order to make it available from any website. Developers, like us, can access these images for our application just by performing a GET operation on a specific API endpoint. In this section, we will look at how to implement Gravatar rather than use the picture provided by the auth service.

Abstracting the avatar URL process

Since we have three different ways of obtaining the avatar URL in our application, we have reached the point where it would be sensible to learn how to abstract the functionality in order to cleanly implement the options. Abstraction refers to a process in which we separate the idea of something from its specific implementation. The `http.Handler` method is a great example of how a handler will be used along with its ins and outs, without being specific about what action is taken by each handler.

In Go, we start to describe our idea of getting an avatar URL by defining an interface. Let's create a new file called `avatar.go` and insert the following code:

```
package main
import (
    "errors"
)
// ErrNoAvatar is the error that is returned when the
// Avatar instance is unable to provide an avatar URL.
var ErrNoAvatarURL = errors.New("chat: Unable to get an avatar URL.")
// Avatar represents types capable of representing
// user profile pictures.
type Avatar interface {
    // GetAvatarURL gets the avatar URL for the specified client,
    // or returns an error if something goes wrong.
    // ErrNoAvatarURL is returned if the object is unable to get
    // a URL for the specified client.
    GetAvatarURL(c *client) (string, error)
}
```

The `Avatar` interface describes the `GetAvatarURL` method that a type must satisfy in order to be able to get avatar URLs. We took the client as an argument so that we know the user for which the URL to be returned. The method returns two arguments: a string (which will be the URL if things go well) and an error in case something goes wrong.

One of the things that could go wrong is simply that one of the specific implementations of Avatar is unable to get the URL. In that case, GetAvatarURL will return the ErrNoAvatarURL error as the second argument. The ErrNoAvatarURL error therefore becomes a part of the interface; it's one of the possible returns from the method and something that users of our code should probably explicitly handle. We mention this in the comments part of the code for the method, which is the only way to communicate such design decisions in Go.



Because the error is initialized immediately using errors.New and stored in the ErrNoAvatarURL variable, only one of these objects will ever be created; passing the pointer of the error as a return is inexpensive. This is unlike Java's checked exceptions which serve a similar purpose where expensive exception objects are created and used as part of the control flow.

The auth service and the avatar's implementation

The first implementation of Avatar we write will replace the existing functionality where we had hardcoded the avatar URL obtained from the auth service. Let's use a **Test-driven Development (TDD)** approach so that we can be sure our code works without having to manually test it. Let's create a new file called `avatar_test.go` in the `chat` folder:

```
package main
import "testing"
func TestAuthAvatar(t *testing.T) {
    var authAvatar AuthAvatar
    client := new(client)
    url, err := authAvatar.GetAvatarURL(client)
    if err != ErrNoAvatarURL {
        t.Error("AuthAvatar.GetAvatarURL should return ErrNoAvatarURL
                when no value present")
    }
    // set a value
    testUrl := "http://url-to-gravatar/"
    client.userData = map[string]interface{}{"avatar_url": testUrl}
    url, err = authAvatar.GetAvatarURL(client)
    if err != nil {
        t.Error("AuthAvatar.GetAvatarURL should return no error
                when value present")
    }
    if url != testUrl {
        t.Error("AuthAvatar.GetAvatarURL should return correct URL")
    }
}
```

This file contains a test for our as-of-yet, nonexistent `AuthAvatar` type's `GetAvatarURL` method. First, it uses a client with no user data and ensures that the `ErrNoAvatarURL` error is returned. After setting a suitable URL, our test calls the method again this time to assert that it returns the correct value. However, building this code fails because the `AuthAvatar` type doesn't exist, so we'll declare `authAvatar` next.

Before we write our implementation, it's worth noticing that we only declare the `authAvatar` variable as the `AuthAvatar` type but never actually assign anything to it so its value remains `nil`. This is not a mistake; we are actually making use of Go's zero-initialization (or default initialization) capabilities. Since there is no state needed for our object (we will pass `client` in as an argument), there is no need to waste time and memory on initializing an instance of it. In Go, it is acceptable to call a method on a `nil` object, provided that the method doesn't try to access a field. When we actually come to writing our implementation, we will look at a way in which we can ensure this is the case.

Let's head back over to `avatar.go` and make our test pass. Add the following code at the bottom of the file:

```
type AuthAvatar struct{}  
var UseAuthAvatar AuthAvatar  
func (AuthAvatar) GetAvatarURL(c *client) (string, error) {  
    if url, ok := c.userData["avatar_url"]; ok {  
        if urlStr, ok := url.(string); ok {  
            return urlStr, nil  
        }  
    }  
    return "", ErrNoAvatarURL  
}
```

Here, we define our `AuthAvatar` type as an empty struct and define the implementation of the `GetAvatarURL` method. We also create a handy variable called `UseAuthAvatar` that has the `AuthAvatar` type but which remains of `nil` value. We can later assign the `UseAuthAvatar` variable to any field looking for an `Avatar` interface type.



The `GetAvatarURL` method we wrote earlier doesn't have a very nice **line of sight**; the happy return is buried within two `if` blocks. See if you can refactor it so that the last line is `return urlStr, nil` and the method exits early if the `avatar_url` field is missing. You can refactor with confidence, since this code is covered by a unit test. For a little more on the rationale behind this kind of refactor, refer to the article at <http://bit.ly/lineofsightgolang>.

Normally, the receiver of a method (the type defined in parentheses before the name) will be assigned to a variable so that it can be accessed in the body of the method. Since, in our case, we assume the object can have `nil` value, we can omit a variable name to tell Go to throw away the reference. This serves as an added reminder to ourselves that we should avoid using it.

The body of our implementation is relatively simple otherwise: we are safely looking for the value of `avatar_url` and ensuring that it is a string before returning it. If anything fails, we return the `ErrNoAvatarURL` error, as defined in the interface.

Let's run the tests by opening a terminal and then navigating to the `chat` folder and typing the following:

```
go test
```

If all is well, our tests will pass and we will have successfully created our first `Avatar` implementation.

Using an implementation

When we use an implementation, we could refer to either the helper variables directly or create our own instance of the interface whenever we need the functionality. However, this would defeat the object of the abstraction. Instead, we use the `Avatar` interface type to indicate where we need the capability.

For our chat application, we will have a single way to obtain an avatar URL per chat room. So, let's update the `room` type so it can hold an `Avatar` object. In `room.go`, add the following field definition to the `room` struct type:

```
// avatar is how avatar information will be obtained.  
avatar Avatar
```

Update the `newRoom` function so that we can pass in an `Avatar` implementation for use; we will just assign this implementation to the new field when we create our `room` instance:

```
// newRoom makes a new room that is ready to go.  
func newRoom/avatar Avatar) *room {  
    return &room{  
        forward: make(chan *message),  
        join:     make(chan *client),  
        leave:   make(chan *client),  
        clients: make(map[*client]bool),  
        tracer:  trace.Off(),  
        avatar:  avatar,
```

```
}
```

Building the project now will highlight the fact that the call to `newRoom` in `main.go` is broken because we have not provided an `Avatar` argument; let's update it by passing in our handy `UseAuthAvatar` variable, as follows:

```
r := newRoom(UseAuthAvatar)
```

We didn't have to create an instance of `AuthAvatar`, so no memory was allocated. In our case, this doesn't result in great saving (since we only have one room for our entire application), but imagine the size of the potential savings if our application has thousands of rooms. The way we named the `UseAuthAvatar` variable means that the preceding code is very easy to read and it also makes our intention obvious.



Thinking about code readability is important when designing interfaces. Consider a method that takes a Boolean input just passing in true or false hides the real meaning if you don't know the argument names. Consider defining a couple of helper constants, as shown in the following short example: `func move(animated bool) { /* ... */ } const Animate = true const DontAnimate = false` Think about which of the following calls to `move` are easier to understand: `move(true)` `move(false)` `move(Animate)` `move(DontAnimate)`

All that is left now is to change `client` to use our new `Avatar` interface. In `client.go`, update the `read` method, as follows:

```
func (c *client) read() {
    defer c.socket.Close()
    for {
        var msg *message
        if err := c.socket.ReadJSON(&msg); err != nil {
            return
        }
        msg.When = time.Now()
        msg.Name = c.userData["name"].(string)
        msg.AvatarURL, _ = c.room.avatar.GetAvatarURL(c)
        c.room.forward <- msg
    }
}
```

Here, we are asking the `avatar` instance in `room` to get the avatar URL for us instead of extracting it from `userData` ourselves.

When you build and run the application, you will notice that (although we have refactored things a little) the behavior and user experience hasn't changed at all. This is because we told our room to use the `AuthAvatar` implementation.

Now let's add another implementation to the room.

The Gravatar implementation

The Gravatar implementation in `Avatar` will do the same job as the `AuthAvatar` implementation, except that it will generate a URL for a profile picture hosted on `https://en.gravatar.com/`. Let's start by adding a test to our `avatar_test.go` file:

```
func TestGravatarAvatar(t *testing.T) {
    var gravatarAvatar GravatarAvatar
    client := new(client)
    client.userData = map[string]interface{}{"email":
        "MyEmailAddress@example.com"}
    url, err := gravatarAvatar.GetAvatarURL(client)
    if err != nil {
        t.Error("GravatarAvatar.GetAvatarURL should not return an error")
    }
    if url != "//www.gravatar.com/avatar/0bc83cb571cd1c50ba6f3e8a78ef1346" {
        t.Errorf("GravatarAvatar.GetAvatarURL wrongly returned %s", url)
    }
}
```

Gravatar uses a hash of the e-mail address to generate a unique ID for each profile picture, so we set up a client and ensure `userData` contains an e-mail address. Next, we call the same `GetAvatarURL` method, but this time on an object that has the `GravatarAvatar` type. We then assert that a correct URL was returned. We already know this is the appropriate URL for the specified e-mail address because it is listed as an example in the Gravatar documentation a great strategy to ensure our code is doing what it should be doing.



Remember that all the source code for this book is available for download from the publishers and has also been published on GitHub. You can save time on building the preceding core by copying and pasting bits and pieces from <https://github.com/matryer/goblueprints>. Hardcoding things such as the base URL is not usually a good idea; we have hardcoded throughout the book to make the code snippets easier to read and more obvious, but you are welcome to extract them as you go along if you like.

Running these tests (with `go test`) obviously causes errors because we haven't defined our types yet. Let's head back to `avatar.go` and add the following code while being sure to import the `io` package:

```
type GravatarAvatar struct{}  
var UseGravatar GravatarAvatar  
func(GravatarAvatar) GetAvatarURL(c *client) (string, error) {  
    if email, ok := c.userData["email"]; ok {  
        if emailStr, ok := email.(string); ok {  
            m := md5.New()  
            io.WriteString(m, strings.ToLower(emailStr))  
            return fmt.Sprintf("//www.gravatar.com/avatar/%x", m.Sum(nil)), nil  
        }  
    }  
    return "", ErrNoAvatarURL  
}
```

We used the same pattern as we did for `AuthAvatar`: we have an empty struct, a helpful `UseGravatar` variable, and the `GetAvatarURL` method implementation itself. In this method, we follow Gravatar's guidelines to generate an MD5 hash from the e-mail address (after we ensured it was lowercase) and append it to the hardcoded base URL using `fmt.Sprintf`.



The preceding method also suffers from a bad line of sight in code. Can you live with it, or would you want to improve the readability somehow?

It is very easy to achieve hashing in Go thanks to the hard work put in by the writers of the Go standard library. The `crypto` package has an impressive array of cryptography and hashing capabilities all very easy to use. In our case, we create a new `md5` hasher and because the hasher implements the `io.Writer` interface, we can use `io.WriteString` to write a string of bytes to it. Calling `Sum` returns the current hash for the bytes written.



You might have noticed that we end up hashing the e-mail address every time we need the avatar URL. This is pretty inefficient, especially at scale, but we should prioritize getting stuff done over optimization. If we need to, we can always come back later and change the way this works.

Running the tests now shows us that our code is working, but we haven't yet included an e-mail address in the auth cookie. We do this by locating the code where we assign to the `authCookieValue` object in `auth.go` and updating it to grab the `Email` value from `Gomniauth`:

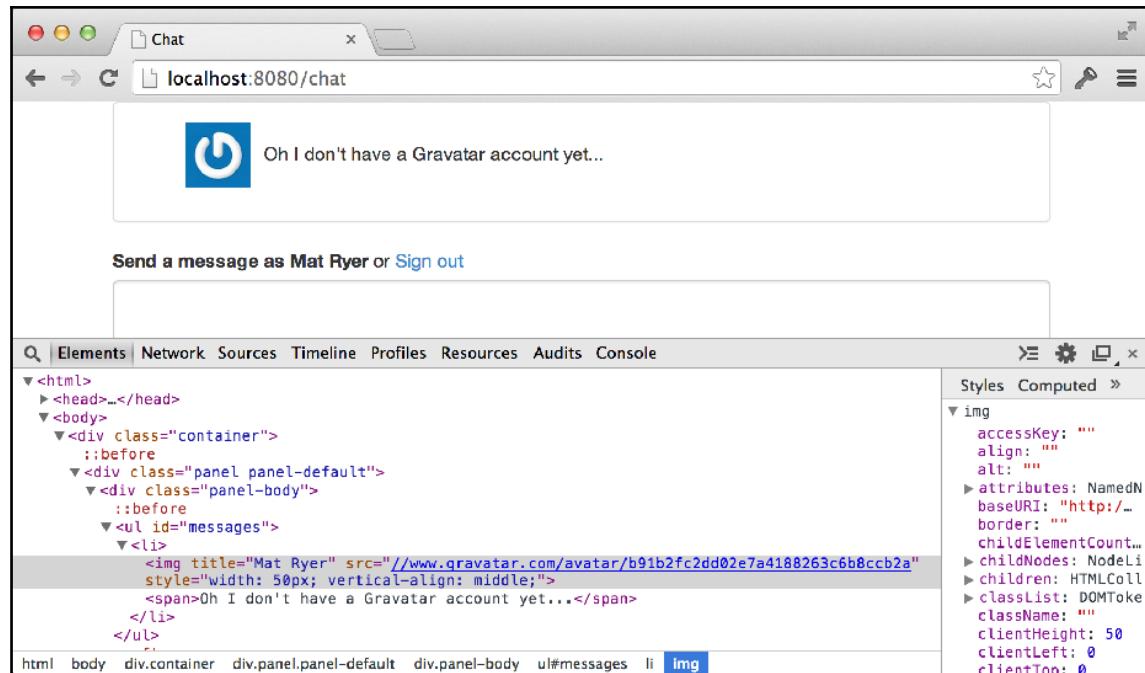
```
authCookieValue := objx.New(map[string]interface{}{  
    "name": user.Name(),  
    "avatar_url": user.AvatarURL(),  
    "email": user.Email(),  
}).MustBase64()
```

The final thing we must do is tell our room to use the Gravatar implementation instead of the `AuthAvatar` implementation. We do this by calling `newRoom` in `main.go` and making the following change:

```
r := newRoom(UseGravatar)
```

Build and run the chat program once again and head to the browser. Remember, since we have changed the information stored in the cookie, we must sign out and sign back in again in order to see our changes take effect.

Assuming you have a different image for your Gravatar account, you will notice that the system is now pulling the image from Gravatar instead of the auth provider. Using your browser's inspector or debug tool will show you that the `src` attribute of the `img` tag has indeed changed:



If you don't have a Gravatar account, you'll most likely see a default placeholder image in place of your profile picture.

Uploading an avatar picture

In the third and final approach of uploading a picture, we will look at how to allow users to upload an image from their local hard drive to use as their profile picture when chatting. The file will then be served to the browsers via a URL. We will need a way to associate a file with a particular user to ensure that we associate the right picture with the corresponding messages.

User identification

In order to uniquely identify our users, we are going to copy Gravatar's approach by hashing their e-mail address and using the resulting string as an identifier. We will store the user ID in the cookie along with the rest of the user-specific data. This will actually have the added benefit of removing the inefficiency associated with continuous hashing from GravatarAuth.

In `auth.go`, replace the code that creates the `authCookieValue` object with the following code:

```
m := md5.New()
io.WriteString(m, strings.ToLower(user.Email()))
userId := fmt.Sprintf("%x", m.Sum(nil))
authCookieValue := objx.New(map[string]interface{}{
    "userid":      userId,
    "name":        user.Name(),
    "avatar_url": user.AvatarURL(),
    "email":       user.Email(),
}).MustBase64()
```

Here, we have hashed the e-mail address and stored the resulting value in the `userid` field at the point at which the user logs in. From now on, we can use this value in our Gravatar code instead of hashing the e-mail address for every message. To do this, first, we update the test by removing the following line from `avatar_test.go`:

```
client.userData = map[string]interface{}{"email":
    "MyEmailAddress@example.com"}
```

We then replace the preceding line with this line:

```
client.userData = map[string]interface{}{"userid":
    "0bc83cb571cd1c50ba6f3e8a78ef1346"}
```

We no longer need to set the `email` field since it is not used; instead, we just have to set an appropriate value to the new `userid` field. However, if you run `go test` in a terminal, you will see this test fail.

To make the test pass, in `avatar.go`, update the `GetAvatarURL` method for the GravatarAuth type:

```
func(GravatarAvatar) GetAvatarURL(c *client) (string, error) {
    if userid, ok := c.userData["userid"]; ok {
        if useridStr, ok := userid.(string); ok {
            return "//www.gravatar.com/avatar/" + useridStr, nil
        }
    }
}
```

```
        }
        return "", ErrNoAvatarURL
    }
```

This won't change the behavior, but it allows us to make an unexpected optimization, which is a great example of why you shouldn't optimize code too early the inefficiencies that you spot early on may not last long enough to warrant the effort required to fix them.

An upload form

If our users are to upload a file as their avatar, they need a way to browse their local hard drive and submit the file to the server. We facilitate this by adding a new template-driven page. In the `chat/templates` folder, create a file called `upload.html`:

```
<html>
  <head>
    <title>Upload</title>
    <link rel="stylesheet"
      href="//netdna.bootstrapcdn.com/bootstrap/3.6.6/css/bootstrap.min.css">
  </head>
  <body>
    <div class="container">
      <div class="page-header">
        <h1>Upload picture</h1>
      </div>
      <form role="form" action="/uploader" enctype="multipart/form-data"
        method="post">
        <input type="hidden" name="userid" value="{{UserData.userid}}"/>
        <div class="form-group">
          <label for="avatarFile">Select file</label>
          <input type="file" name="avatarFile"/>
        </div>
        <input type="submit" value="Upload" class="btn" />
      </form>
    </div>
  </body>
</html>
```

We used Bootstrap again to make our page look nice and also to make it fit in with the other pages. However, the key point to note here is the HTML form that will provide the user interface required to upload files. The action points to `/uploader`, the handler for which we have yet to implement, and the `enctype` attribute must be `multipart/form-data` so that the browser can transmit binary data over HTTP. Then, there is an `input` element of the type `file`, which will contain a reference to the file we want to upload. Also, note that we have included the `userid` value from the `UserData` map as a hidden input this will tell us which user is uploading a file. It is important that the `name` attributes be correct, as this is how we will refer to the data when we implement our handler on the server.

Let's now map the new template to the `/upload` path in `main.go`:

```
http.Handle("/upload", &templateHandler{filename: "upload.html"})
```

Handling the upload

When the user clicks on **Upload** after selecting a file, the browser will send the data for the file as well as the user ID to `/uploader`, but right now, that data doesn't actually go anywhere. We will implement a new `HandlerFunc` interface that is capable of receiving the file, reading the bytes that are streamed through the connection, and saving it as a new file on the server. In the `chat` folder, let's create a new folder called `avatars` this is where we will save the avatar image files.

Next, create a new file called `upload.go` and insert the following code make sure that you add the appropriate package name and imports (which are `ioutil`, `net/http`, `io`, and `path`):

```
func uploaderHandler(w http.ResponseWriter, req *http.Request) {
    userId := req.FormValue("userid")
    file, header, err := req.FormFile("avatarFile")
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    data, err := ioutil.ReadAll(file)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    filename := path.Join("avatars", userId+path.Ext(header.Filename))
    err = ioutil.WriteFile(filename, data, 0777)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
    }
}
```

```
        return
    }
    io.WriteString(w, "Successful")
}
```

Here, first `uploaderHandler` uses the `FormValue` method in `http.Request` to get the user ID that we placed in the hidden input in our HTML form. Then, it gets an `io.Reader` type capable of reading the uploaded bytes by calling `req.FormFile`, which returns three arguments. The first argument represents the file itself with the `multipart.File` interface type, which is also `io.Reader`. The second is a `multipart.FileHeader` object that contains the metadata about the file, such as the filename. And finally, the third argument is an error that we hope will have a `nil` value.

What do we mean when we say that the `multipart.File` interface type is also `io.Reader`? Well, a quick glance at the documentation at <http://golang.org/pkg/mime/multipart/#File> makes it clear that the type is actually just a wrapper interface for a few other more general interfaces. This means that a `multipart.File` type can be passed to methods that require `io.Reader`, since any object that implements `multipart.File` must, therefore, implement `io.Reader`.



Embedding standard library interfaces, such as the wrapper, to describe new concepts is a great way to make sure your code works in as many contexts as possible. Similarly, you should try to write code that uses the simplest interface type you can find, ideally from the standard library. For example, if you wrote a method that needed you to read the contents of a file, you could ask the user to provide an argument of the type `multipart.File`. However, if you ask for `io.Reader` instead, your code will become significantly more flexible because any type that has the appropriate `Read` method can be passed in, which includes user-defined types as well.

The `ioutil.ReadAll` method will just keep reading from the specified `io.Reader` interface until all of the bytes have been received, so this is where we actually receive the stream of bytes from the client. We then use `path.Join` and `path.Ext` to build a new filename using `userid` and copy the extension from the original filename that we can get from `multipart.FileHeader`.

We then use the `ioutil.WriteFile` method to create a new file in the `avatars` folder. We use `userid` in the filename to associate the image with the correct user, much in the same way as Gravatar does. The `0777` value specifies that the new file we create should have complete file permissions, which is a good default setting if you're not sure what other permissions should be set.

If an error occurs at any stage, our code will write it out to the response along with a 500 status code (since we specify `http.StatusInternalServerError`), which will help us debug it, or it will write **Successful** if everything went well.

In order to map this new handler function to `/uploader`, we need to head back to `main.go` and add the following line to `func main`:

```
http.HandleFunc("/uploader", uploaderHandler)
```

Now build and run the application and remember to log out and log back in again in order to give our code a chance to upload the `auth` cookie:

```
go build -o chat
./chat -host=:8080
```

Open `http://localhost:8080/upload` and click on **Choose File**, and then select a file from your hard drive and click on **Upload**. Navigate to your `chat/avatars` folder and you will notice that the file was indeed uploaded and renamed to the value of your `userid` field.

Serving the images

Now that we have a place to keep our user's avatar images on the server, we need a way to make them accessible to the browser. We do this using the `net/http` package's built-in file server. In `main.go`, add the following code:

```
http.Handle("/avatars/",
    http.StripPrefix("/avatars/",
        http.FileServer(http.Dir("./avatars"))))
```

This is actually a single line of code that has been broken up to improve readability. The `http.Handle` call should feel familiar, as we are specifying that we want to map the `/avatars/` path with the specified handler this is where things get interesting. Both `http.StripPrefix` and `http.FileServer` return `http.Handler`, and they make use of the wrapping pattern we learned about in the previous chapter. The `StripPrefix` function takes `http.Handler` in, modifies the path by removing the specified prefix, and passes the functionality onto an inner handler. In our case, the inner handler is an `http.FileServer` handler that will simply serve static files, provide index listings, and generate the 404 Not Found error if it cannot find the file. The `http.Dir` function allows us to specify which folder we want to expose publicly.



If we didn't strip the `/avatars/` prefix from the requests with `http.StripPrefix`, the file server would look for another folder called `avatars` inside the actual `avatars` folder, that is, `/avatars/avatars/filename` instead of `/avatars/filename`.

Let's build the program and run it before opening `http://localhost:8080/avatars/` in a browser. You'll notice that the file server has generated a listing of the files inside our `avatars` folder. Clicking on a file will either download the file, or in the case of an image, simply display it. If you haven't done this already, go to `http://localhost:8080/upload` and upload a picture, and then head back to the listing page and click on it to see it in the browser.

The Avatar implementation for local files

The final step in making filesystem avatars work is writing an implementation of our `Avatar` interface that generates URLs that point to the filesystem endpoint we created in the previous section.

Let's add a test function to our `avatar_test.go` file:

```
func TestFileSystemAvatar(t *testing.T) {
    filename := filepath.Join("avatars", "abc.jpg")
    ioutil.WriteFile(filename, []byte{}, 0777)
    defer os.Remove(filename)
    var fileSystemAvatar FileSystemAvatar
    client := new(client)
    client.userData = map[string]interface{}{"userid": "abc"}
    url, err := fileSystemAvatar.GetAvatarURL(client)
    if err != nil {
        t.Error("FileSystemAvatar.GetAvatarURL should not return an error")
    }
    if url != "/avatars/abc.jpg" {
        t.Errorf("FileSystemAvatar.GetAvatarURL wrongly returned %s", url)
    }
}
```

This test is similar to, but slightly more involved than, the `GravatarAvatar` test because we are also creating a test file in our `avatars` folder and deleting it afterwards.



Even if our test code panics, the deferred functions will still be called. So regardless of what happens, our test code will clean up after itself.

The rest of the test is simple: we set a `userid` field in `client.userData` and call `GetAvatarURL` to ensure we get the right value back. Of course, running this test will fail, so let's go and add the following code in order to make it pass in `avatar.go`:

```
type FileSystemAvatar struct{}  
var UseFileSystemAvatar FileSystemAvatar  
func (FileSystemAvatar) GetAvatarURL(c *client) (string, error) {  
    if userid, ok := c.userData["userid"]; ok {  
        if useridStr, ok := userid.(string); ok {  
            return "/avatars/" + useridStr + ".jpg", nil  
        }  
    }  
    return "", ErrNoAvatarURL  
}
```

As you can see here, in order to generate the correct URL, we simply get the `userid` value and build the final string by adding the appropriate segments together. You may have noticed that we have hardcoded the file extension to `.jpg`, which means that the initial version of our chat application will only support JPEGs.



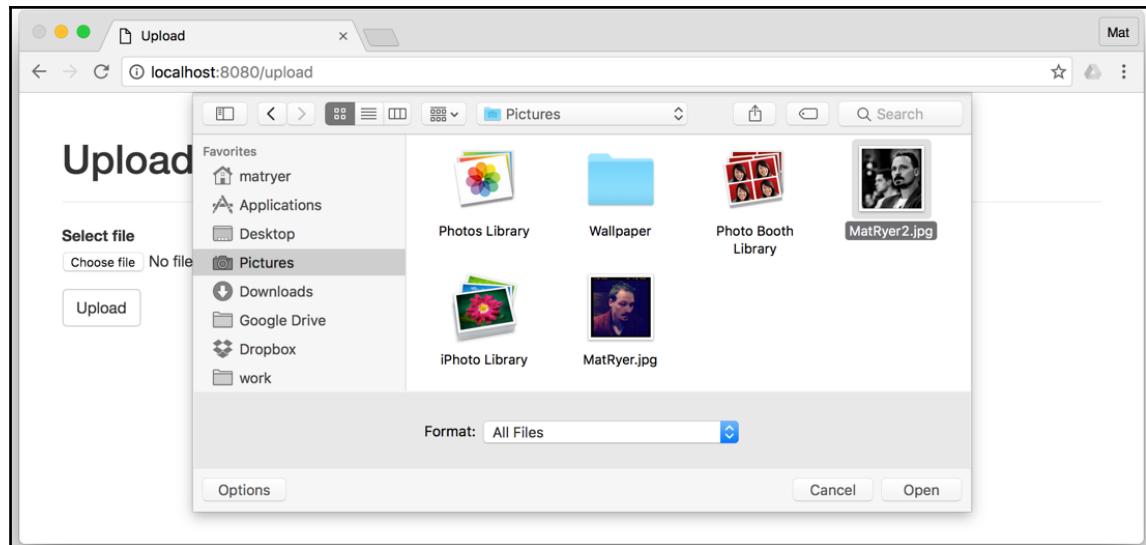
Supporting only JPEGs might seem like a half-baked solution, but following Agile methodologies, this is perfectly fine; after all, custom JPEG profile pictures are better than no custom profile pictures at all.

Let's look at our new code in action by updating `main.go` to use our new Avatar implementation:

```
r := newRoom(UseFileSystemAvatar)
```

Now build and run the application as usual and go to `http://localhost:8080/upload` and use a web form to upload a JPEG image to use as your profile picture. To make sure it's working correctly, choose a unique image that isn't your Gravatar picture or the image from the auth service. Once you see the successful message after clicking on **Upload**, go to `http://localhost:8080/chat` and post a message. You will notice that the application has indeed used the profile picture that you uploaded.

To change your profile picture, go back to the `/upload` page and upload a different picture, and then jump back to the `/chat` page and post more messages.



Supporting different file types

To support different file types, we have to make our `GetAvatarURL` method for the `FileSystemAvatar` type a little smarter.

Instead of just blindly building the string, we will use the very important `ioutil.ReadDir` method to get a listing of the files. The listing also includes directories so we will use the `IsDir` method to determine whether we should skip it or not.

We will then check whether each file matches the `userid` field (remember that we named our files in this way) by a call to `path.Match`. If the filename matches the `userid` field, then we have found the file for that user and we return the path. If anything goes wrong or if we can't find the file, we return the `ErrNoAvatarURL` error as usual.

Update the appropriate method in `avatar.go` with the following code:

```
func (FileSystemAvatar) GetAvatarURL(c *client) (string, error) {
    if userid, ok := c.userData["userid"]; ok {
        if useridStr, ok := userid.(string); ok {
            files, err := ioutil.ReadDir("avatars")
            if err != nil {
                return "", ErrNoAvatarURL
            }
            for _, file := range files {
                if file.IsDir() {
```

```
        continue
    }
    if match, _ := path.Match(useridStr+"*", file.Name());
    match {
        return "/avatars/" + file.Name(), nil
    }
}
}
return "", ErrNoAvatarURL
}
```

Delete all the files in the `avatar` folder to prevent confusion and rebuild the program. This time, upload an image of a different type and note that our application has no difficulty handling it.

Refactoring and optimizing our code

When we look back at how our `Avatar` type is used, you will notice that every time someone sends a message, the application makes a call to `GetAvatarURL`. In our latest implementation, each time the method is called, we iterate over all the files in the `avatars` folder. For a particularly chatty user, this could mean that we end up iterating over and over again many times a minute. This is an obvious waste of resources and would, at some point very soon, become a scaling problem.

Instead of getting the avatar URL for every message, we should get it only once when the user first logs in and cache it in the auth cookie. Unfortunately, our `Avatar` interface type requires that we pass in a `client` object to the `GetAvatarURL` method and we do not have such an object at the point at which we are authenticating the user.



So did we make a mistake when we designed our `Avatar` interface? While this is a natural conclusion to come to, in fact we did the right thing. We designed the solution with the best information we had available at the time and therefore had a working chat application much sooner than if we'd tried to design for every possible future case. Software evolves and almost always changes during the development process and will definitely change throughout the lifetime of the code.

Replacing concrete types with interfaces

We have concluded that our `GetAvatarURL` method depends on a type that is not available to us at the point we need it, so what would be a good alternative? We could pass each required field as a separate argument, but this would make our interface brittle, since as soon as an `Avatar` implementation needs a new piece of information, we'd have to change the method signature. Instead, we will create a new type that will encapsulate the information our `Avatar` implementations need while conceptually remaining decoupled from our specific case.

In `auth.go`, add the following code to the top of the page (underneath the `package` keyword, of course):

```
import gomniauthcommon "github.com/stretchr/gomniauth/common"
type ChatUser interface {
    UniqueID() string
    AvatarURL() string
}
type chatUser struct {
    gomniauthcommon.User
    uniqueID string
}
func (u chatUser) UniqueID() string {
    return u.uniqueID
}
```

Here, the `import` statement imported the `common` package from Gomniauth and, at the same time, gave it a specific name through which it will be accessed: `gomniauthcommon`. This isn't entirely necessary since we have no package name conflicts. However, it makes the code easier to understand.

In the preceding code snippet, we also defined a new interface type called `ChatUser`, which exposes the information needed in order for our `Avatar` implementations to generate the correct URLs. Then, we defined an actual implementation called `chatUser` (notice the lowercase starting letter) that implements the interface. It also makes use of a very interesting feature in Go: type embedding. We actually embedded the `gomniauth/common.User` interface type, which means that our `struct` interface implements the interface automatically.

You may have noticed that we only actually implemented one of the two required methods to satisfy our ChatUser interface. We got away with this because the Gomniauth User interface happens to define the same AvatarURL method. In practice, when we instantiate our chatUser struct provided we set an appropriate value for the implied Gomniauth User field our object implements both Gomniauth's User interface and our own ChatUser interface at the same time.

Changing interfaces in a test-driven way

Before we can use our new type, we must update the Avatar interface and appropriate implementations to make use of it. As we will follow TDD practices, we are going to first make these changes in our test file, see the compiler errors when we try to build our code, and see failing tests once we fix those errors before finally making the tests pass.

Open `avatar_test.go` and replace `TestAuthAvatar` with the following code:

```
func TestAuthAvatar(t *testing.T) {
    var authAvatar AuthAvatar
    testUser := &gomniauthtest.TestUser{}
    testUser.On("AvatarURL").Return("", ErrNoAvatarURL)
    testChatUser := &chatUser{User: testUser}
    url, err := authAvatar.GetAvatarURL(testChatUser)
    if err != ErrNoAvatarURL {
        t.Error("AuthAvatar.GetAvatarURL should return ErrNoAvatarURL
                when no value present")
    }
    testUrl := "http://url-to-gravatar/"
    testUser = &gomniauthtest.TestUser{}
    testChatUser.User = testUser
    testUser.On("AvatarURL").Return(testUrl, nil)
    url, err = authAvatar.GetAvatarURL(testChatUser)
    if err != nil {
        t.Error("AuthAvatar.GetAvatarURL should return no error
                when value present")
    }
    if url != testUrl {
        t.Error("AuthAvatar.GetAvatarURL should return correct URL")
    }
}
```



You will also need to import the `gomniauth/test` package as `gomniauthtest`, like we did in the last section.

Using our new interface before we have defined it is a good way to check the sanity of our thinking, which is another advantage of practicing TDD. In this new test, we create `TestUser` provided by Gomniauth and embed it into a `chatUser` type. We then pass the new `chatUser` type into our `GetAvatarURL` calls and make the same assertions about output as we always have done.



Gomniauth's `TestUser` type is interesting as it makes use of the `Testify` package's mocking capabilities. Refer to <https://github.com/stretchr/testify> for more information. The `On` and `Return` methods allow us to tell `TestUser` what to do when specific methods are called. In the first case, we tell the `AvatarURL` method to return the error, and in the second case, we ask it to return the `testUrl` value, which simulates the two possible outcomes we are covering in this test.

Updating the other two tests is much simpler because they rely only on the `UniqueID` method, the value of which we can control directly.

Replace the other two tests in `avatar_test.go` with the following code:

```
func TestGravatarAvatar(t *testing.T) {
    var gravatarAvatar GravatarAvatar
    user := &chatUser{uniqueID: "abc"}
    url, err := gravatarAvatar.GetAvatarURL(user)
    if err != nil {
        t.Error("GravatarAvatar.GetAvatarURL should not return an error")
    }
    if url != "//www.gravatar.com/avatar/abc" {
        t.Errorf("GravatarAvatar.GetAvatarURL wrongly returned %s", url)
    }
}
func TestFileSystemAvatar(t *testing.T) {
    // make a test avatar file
    filename := path.Join("avatars", "abc.jpg")
    ioutil.WriteFile(filename, []byte{}, 0777)
    defer func() { os.Remove(filename) }()
    var fileSystemAvatar FileSystemAvatar
    user := &chatUser{uniqueID: "abc"}
    url, err := fileSystemAvatar.GetAvatarURL(user)
    if err != nil {
        t.Error("FileSystemAvatar.GetAvatarURL should not return an error")
    }
    if url != "/avatars/abc.jpg" {
        t.Errorf("FileSystemAvatar.GetAvatarURL wrongly returned %s", url)
    }
}
```

Of course, this test code won't even compile because we are yet to update our Avatar interface. In `avatar.go`, update the `GetAvatarURL` signature in the `Avatar` interface type to take a `ChatUser` type rather than a `client` type:

```
GetAvatarURL(ChatUser) (string, error)
```



Note that we are using the `ChatUser` interface (with the starting letter in uppercase) rather than our internal `chatUser` implementation struct after all, we want to be flexible about the types our `GetAvatarURL` methods accept.

Trying to build this will reveal that we now have broken implementations because all the `GetAvatarURL` methods are still asking for a `client` object.

Fixing the existing implementations

Changing an interface like the one we have is a good way to automatically find the parts of our code that have been affected because they will cause compiler errors. Of course, if we were writing a package that other people would use, we would have to be far stricter about changing the interfaces like this, but we haven't released our v1 yet, so it's fine.

We are now going to update the three implementation signatures to satisfy the new interface and change the method bodies to make use of the new type. Replace the implementation for `FileSystemAvatar` with the following:

```
func (FileSystemAvatar) GetAvatarURL(u ChatUser) (string, error) {
    if files, err := ioutil.ReadDir("avatars"); err == nil {
        for _, file := range files {
            if file.IsDir() {
                continue
            }
            if match, _ := path.Match(u.UniqueID()+"*", file.Name());
            match {
                return "/avatars/" + file.Name(), nil
            }
        }
    }
    return "", ErrNoAvatarURL
}
```

The key change here is that we no longer access the `userData` field on the client, and just call `UniqueID` directly on the `ChatUser` interface instead.

Next, we update the `AuthAvatar` implementation with the following code:

```
func (AuthAvatar) GetAvatarURL(u ChatUser) (string, error) {
    url := u.AvatarURL()
    if len(url) == 0 {
        return "", ErrNoAvatarURL
    }
    return url, nil
}
```

Our new design proves to be much simpler, it's always a good thing if we can reduce the amount of code required. The preceding code makes a call to get the `AvatarURL` value, and provided it isn't empty, we return it; otherwise, we return the `ErrNoAvatarURL` error.



Note how the expected flow of the code is indented to one level, while error cases are nested inside `if` blocks. While you can't stick to this practice 100% of the time, it's a worthwhile endeavor. Being able to quickly scan the code (when reading it) to see the normal flow of execution down a single column allows you to understand the code much quicker. Compare this to code that has lots of `if...else` nested blocks, which takes a lot more unpicking to understand.

Finally, update the `GravatarAvatar` implementation:

```
func (GravatarAvatar) GetAvatarURL(u ChatUser) (string, error) {
    return "//www.gravatar.com/avatar/" + u.UniqueID(), nil
}
```

Global variables versus fields

So far, we have assigned the `Avatar` implementation to the `room` type, which enables us to use different avatars for different rooms. However, this has exposed an issue: when our users sign in, there is no concept of which room they are headed to so we cannot know which `Avatar` implementation to use. Because our application only supports a single room, we are going to look at another approach to select implementations: the use of global variables.

A global variable is simply a variable that is defined outside any type definition and is accessible from every part of the package (and from outside the package if it's exported). For a simple configuration, such as which type of Avatar implementation to use, global variables are an easy and simple solution. Underneath the `import` statements in `main.go`, add the following line:

```
// set the active Avatar implementation
var avatars Avatar = UseFileSystemAvatar
```

This defines `avatars` as a global variable that we can use when we need to get the avatar URL for a particular user.

Implementing our new design

We need to change the code that calls `GetAvatarURL` for every message to just access the value that we put into the `userData` cache (via the `auth` cookie). Change the line where `msg.AvatarURL` is assigned, as follows:

```
if avatarUrl, ok := c.userData["avatar_url"]; ok {
    msg.AvatarURL = avatarUrl.(string)
}
```

Find the code inside `loginHandler` in `auth.go` where we call `provider.GetUser` and replace it, down to where we set the `authCookieValue` object, with the following code:

```
user, err := provider.GetUser(creds)
if err != nil {
    log.Fatalln("Error when trying to get user from", provider, "-", err)
}
chatUser := &chatUser{User: user}
m := md5.New()
io.WriteString(m, strings.ToLower(user.Email()))
chatUser.uniqueID = fmt.Sprintf("%x", m.Sum(nil))
avatarURL, err := avatars.GetAvatarURL(chatUser)
if err != nil {
    log.Fatalln("Error when trying to GetAvatarURL", "-", err)
}
```

Here, we created a new `chatUser` variable while setting the `User` field (which represents the embedded interface) to the `User` value returned from `Gomniauth`. We then saved the `userid` MD5 hash to the `uniqueID` field.

The call to `avatars.GetAvatarURL` is where all of our hard work has paid off, as we now get the avatar URL for the user far earlier in the process. Update the `authCookieValue` line in `auth.go` to cache the avatar URL in the cookie and remove the e-mail address since it is no longer required:

```
authCookieValue := objx.New(map[string]interface{}){
    "userid":      chatUser.uniqueID,
    "name":        user.Name(),
    "avatar_url":  avatarURL,
}) .MustBase64()
```

However expensive the work the `Avatar` implementation needs to do, such as iterating over files on the filesystem, it is mitigated by the fact that the implementation only does so when the user first logs in and not every time they send a message.

Tidying up and testing

Finally, we get to snip away at some of the fat that has accumulated during our refactoring process.

Since we no longer store the `Avatar` implementation in `room`, let's remove the field and all references to it from the type. In `room.go`, delete the `avatar Avatar` definition from the `room` struct and update the `newRoom` method:

```
func newRoom() *room {
    return &room{
        forward: make(chan *message),
        join:    make(chan *client),
        leave:   make(chan *client),
        clients: make(map[*client]bool),
        tracer:  trace.Off(),
    }
}
```

Remember to use the compiler as your to-do list where possible, and follow the errors to find where you have impacted other code.



In `main.go`, remove the parameter passed into the `newRoom` function call since we are using our global variable instead of this one.

After this exercise, the end user experience remains unchanged. Usually when refactoring the code, it is the internals that are modified while the public-facing interface remains stable and unchanged. As you go, remember to re-run the unit tests to make sure you don't break anything as you evolve the code.



It's usually a good idea to run tools such as `golint` and `go vet` against your code as well in order to make sure it follows good practices and doesn't contain any Go faux pas, such as missing comments or badly named functions. There are a few deliberately left in for you to fix yourself.

Combining all three implementations

To close this chapter with a bang, we will implement a mechanism in which each `Avatar` implementation takes a turn in trying to get a URL for a user. If the first implementation returns the `ErrNoAvatarURL` error, we will try the next and so on until we find a useable value.

In `avatar.go`, underneath the `Avatar` type, add the following type definition:

```
type TryAvatars []Avatar
```

The `TryAvatars` type is simply a slice of `Avatar` objects that we are free to add methods to. Let's add the following `GetAvatarURL` method:

```
func (a TryAvatars) GetAvatarURL(u ChatUser) (string, error) {
    for _, avatar := range a {
        if url, err := avatar.GetAvatarURL(u); err == nil {
            return url, nil
        }
    }
    return "", ErrNoAvatarURL
}
```

This means that `TryAvatars` is now a valid `Avatar` implementation and can be used in place of any specific implementation. In the preceding method, we iterated over the slice of `Avatar` objects in an order, calling `GetAvatarURL` for each one. If no error is returned, we return the URL; otherwise, we carry on looking. Finally, if we are unable to find a value, we just return `ErrNoAvatarURL` as per the interface design.

Update the `avatars` global variable in `main.go` to use our new implementation:

```
var avatars Avatar = TryAvatars{  
    UseFileSystemAvatar,  
    UseAuthAvatar,  
    UseGravatar}
```

Here, we created a new instance of our `TryAvatars` slice type while putting the other `Avatar` implementations inside it. The order matters since it iterates over the objects in the order in which they appear in the slice. So, first our code will check whether the user has uploaded a picture; if they haven't, the code will check whether the auth service has a picture for us to use. If the approaches fail, a Gravatar URL will be generated, which in the worst case (for example, if the user hasn't added a Gravatar picture) will render a default placeholder image.

To see our new functionality in action, perform the following steps:

1. Build and rerun the application:

```
go build -o chat  
./chat -host=:8080
```

2. Log out by visiting `http://localhost:8080/logout`.
3. Delete all the pictures from the `avatars` folder.
4. Log back in by navigating to `http://localhost:8080/chat`.
5. Send some messages and take note of your profile picture.
6. Visit `http://localhost:8080/upload` and upload a new profile picture.
7. Log out again and log back in as you did earlier.
8. Send some more messages and note that your profile picture has been updated.

Summary

In this chapter, we added three different implementations of profile pictures to our chat application. First, we asked the auth service to provide a URL for us to use. We did this using Gomniauth's abstraction of the user resource data, which we then included as part of the user interface every time a user would send a message. Using Go's zero (or default) initialization, we were able to refer to different implementations of our `Avatar` interface without actually creating any instances.

We stored data in a cookie for when the user would log in. Given the fact that cookies persist between builds of our code, we added a handy logout feature to help us validate our changes, which we also exposed to our users so that they could log out too. Other small changes to the code and the inclusion of Bootstrap on our chat page dramatically improved the look and feel of our application.

We used MD5 hashing in Go to implement the <https://en.gravatar.com/> API by hashing the e-mail address that the auth service provided. If the e-mail address is not known to Gravatar, they will deliver a nice default placeholder image for us, which means our user interface will never be broken due to missing images.

We then built and completed an upload form and associated the server functionality that saved uploaded pictures in the `avatars` folder. We saw how to expose the saved uploaded pictures to users via the standard library's `http.FileServer` handler. As this introduced inefficiencies in our design by causing too much filesystem access, we refactored our solution with the help of our unit tests. By moving the `GetAvatarURL` call to the point at which users log in rather than every time a message is sent, we made our code significantly more scalable.

Our special `ErrNoAvatarURL` error type was used as part of our interface design in order to allow us to inform the calling code when it was not possible to obtain an appropriate URL. This became particularly useful when we created our `Avatars` slice type. By implementing the `Avatar` interface on a slice of `Avatar` types, we were able to create a new implementation that took turns trying to get a valid URL from each of the different options available, starting with the filesystem, then the auth service, and finally Gravatar. We achieved this with zero impact on how the user would interact with the interface. If an implementation returned `ErrNoAvatarURL`, we tried the next one.

Our chat application is ready to go live, so we can invite our friends and have a real conversation. But first, we need to choose a domain name to host it at, something we will look at in the next chapter.

4

Command-Line Tools to Find Domain Names

The chat application we've built so far is ready to take the world by storm but not before we give it a home on the Internet. Before we invite our friends to join the conversation, we need to pick a valid, catchy, and available domain name, which we can point to the server running our Go code. Instead of sitting in front of our favorite domain name provider for hours on end trying different names, we are going to develop a few command-line tools that will help us find the right one. As we do so, we will see how the Go standard library allows us to interface with the terminal and other executing applications; we'll also explore some patterns and practices to build command-line programs.

In this chapter, you will learn:

- How to build complete command-line applications with as little as a single code file
- How to ensure that the tools we build can be composed with other tools using standard streams
- How to interact with a simple third-party JSON RESTful API
- How to utilize the standard in and out pipes in Go code
- How to read from a streaming source, one line at a time
- How to build a WHOIS client to look up domain information
- How to store and use sensitive or deployment-specific information in environment variables

Pipe design for command-line tools

We are going to build a series of command-line tools that use the standard streams (`stdin` and `stdout`) to communicate with the user and with other tools. Each tool will take an input line by line via the standard input pipe, process it in some way, and then print the output line by line to the standard out pipe for the next tool or user.

By default, the standard input is connected to the user's keyboard, and the standard output is printed to the terminal from where the command was run; however, both can be redirected using **redirection metacharacters**. It's possible to throw the output away by redirecting it to `NUL` on Windows or `/dev/null` on Unix machines, or redirecting it to a file that will cause the output to be saved to a disk. Alternatively, you can pipe (using the `|` pipe character) the output of one program to the input of another; it is this feature that we will make use of in order to connect our various tools together. For example, you could pipe the output from one program to the input of another program in a terminal using this code:

```
echo -n "Hello" | md5
```

The output of the `echo` command will be the string `Hello` (without the quotes), which is then **piped** to the `md5` command; this command will in turn calculate the MD5 hash of `Hello`:

```
8b1a9953c4611296a827abf8c47804d7
```

Our tools will work with lines of strings where each line (separated by a linefeed character) represents one string. When run without any pipe redirection, we will be able to interact directly with the programs using the default in and out, which will be useful when testing and debugging our code.

Five simple programs

In this chapter, we will build five small programs that we will combine at the end. The key features of the programs are as follows:

- **Sprinkle**: This program will add some web-friendly sprinkle words to increase the chances of finding the available domain names.
- **Domainify**: This program will ensure words are acceptable for a domain name by removing unacceptable characters. Once this is done, it will replace spaces with hyphens and add an appropriate top-level domain (such as `.com` and `.net`) to the end.

- **Coolify:** This program will change a boring old normal word to Web 2.0 by fiddling around with vowels.
- **Synonyms:** This program will use a third-party API to find synonyms.
- **Available:** This program will use a third-party API to find synonyms. Available: This program will check to see whether the domain is available or not using an appropriate WHOIS server.

Five programs might seem like a lot for one chapter, but don't forget how small entire programs can be in Go.

Sprinkle

Our first program augments the incoming words with some sugar terms in order to improve the odds of finding names that are available. Many companies use this approach to keep the core messaging consistent while being able to afford the .com domain. For example, if we pass in the word chat, it might pass out chatapp; alternatively, if we pass in talk, we may get back talk time.

Go's `math/rand` package allows us to break away from the predictability of computers. It gives our program the appearance of intelligence by introducing elements of chance into its decision making.

To make our Sprinkle program work, we will:

- Define an array of transformations, using a special constant to indicate where the original word will appear
- Use the `bufio` package to scan the input from `stdin` and `fmt.Println` in order to write the output to `stdout`
- Use the `math/rand` package to randomly select a transformation to apply



All our programs will reside in the `$GOPATH/src` directory. For example, if your `GOPATH` is `~/Work/projects/go`, you would create your program folders in the `~/Work/projects/go/src` folder.

In the `$GOPATH/src` directory, create a new folder called `sprinkle` and add a `main.go` file containing the following code:

```
package main
import (
    "bufio"
```

```
"fmt"
"math/rand"
"os"
"strings"
"time"
)
const otherWord = "*"
var transforms = []string{
otherWord,
otherWord + "app",
otherWord + "site",
otherWord + "time",
"get" + otherWord,
"go" + otherWord,
"lets " + otherWord,
otherWord + "hq",
}
func main() {
rand.Seed(time.Now().UTC().UnixNano())
s := bufio.NewScanner(os.Stdin)
for s.Scan() {
t := transforms[rand.Intn(len(transforms))]
fmt.Println(strings.Replace(t, otherWord, s.Text(), -1))
}
}
```

From now on, it is assumed that you will sort out the appropriate `import` statements yourself. If you need assistance, refer to the tips provided in [Appendix, Good Practices for a Stable Go Environment](#).

The preceding code represents our complete Sprinkle program. It defines three things: a constant, a variable, and the obligatory `main` function, which serves as the entry point to Sprinkle. The `otherWord` constant string is a helpful token that allows us to specify where the original word should occur in each of our possible transformations. It lets us write code, such as `otherWord+"extra"`, which makes it clear that in this particular case, we want to add the word "extra" to the end of the original word.

The possible transformations are stored in the `transforms` variable that we declare as a slice of strings. In the preceding code, we defined a few different transformations, such as adding `app` to the end of a word or `lets` before it. Feel free to add some more; the more creative, the better.

In the `main` function, the first thing we do is use the current time as a random seed. Computers can't actually generate random numbers, but changing the seed number of random algorithms gives the illusion that it can. We use the current time in nanoseconds because it's different each time the program is run (provided the system clock isn't being reset before each run). If we skip this step, the numbers generated by the `math/rand` package would be deterministic; they'd be the same every time we run the program.

We then create a `bufio.Scanner` object (by calling `bufio.NewScanner`) and tell it to read the input from `os.Stdin`, which represents the standard input stream. This will be a common pattern in our five programs since we are always going to read from the standard *in* and write to the standard *out*.



The `bufio.Scanner` object actually takes `io.Reader` as its input source, so there is a wide range of types that we could use here. If you were writing unit tests for this code, you could specify your own `io.Reader` for the scanner to read from, removing the need for you to worry about simulating the standard input stream.

As the default case, the scanner allows us to read blocks of bytes separated by defined delimiters, such as carriage return and linefeed characters. We can specify our own split function for the scanner or use one of the options built in the standard library. For example, there is `bufio.ScanWords`, which scans individual words by breaking on whitespace rather than linefeeds. Since our design specifies that each line must contain a word (or a short phrase), the default line-by-line setting is ideal.

A call to the `Scan` method tells the scanner to read the next block of bytes (the next line) from the input, and then it returns a `bool` value indicating whether it found anything or not. This is how we are able to use it as the condition for the `for` loop. While there is content to work on, `Scan` returns `true` and the body of the `for` loop is executed; when `Scan` reaches the end of the input, it returns `false`, and the loop is broken. The bytes that are selected are stored in the `Bytes` method of the scanner, and the handy `Text` method that we use converts the `[]byte` slice into a string for us.

Inside the `for` loop (so for each line of input), we use `rand.Intn` to select a random item from the `transforms` slice and use `strings.Replace` to insert the original word where the `otherWord` string appears. Finally, we use `fmt.Println` to print the output to the default standard output stream.



The `math/rand` package provides insecure random numbers. If you want to write code that utilizes random numbers for security purposes, you must use the `crypto/rand` package instead.

Let's build our program and play with it:

```
go build -o sprinkle  
./sprinkle
```

Once the program starts running, it will use the default behavior to read the user input from the terminal. It uses the default behavior because we haven't piped in any content or specified a source for it to read from. Type `chat` and hit return. The scanner in our code notices the linefeed character at the end of the word and runs the code that transforms it, outputting the result. For example, if you type `chat` a few times, you would see the following output:

```
chat  
go chat  
chat  
lets chat  
chat  
chat app
```

Sprinkle never exits (meaning the `Scan` method never returns `false` to break the loop) because the terminal is still running; in normal execution, the `in` pipe will be closed by whatever program is generating the input. To stop the program, hit `Ctrl + C`.

Before we move on, let's try to run Sprinkle, specifying a different input source. We are going to use the `echo` command to generate some content and pipe it to our Sprinkle program using the pipe character:

```
echo "chat" | ./sprinkle
```

The program will randomly transform the word, print it out, and exit since the `echo` command generates only one line of input before terminating and closing the pipe.

We have successfully completed our first program, which has a very simple but useful function, as we will see.



As an extra assignment, rather than hardcoding the `transformations` array as we have done, see whether you can externalize it via flags or store them in a text file or database.

Domainify

Some of the words that output from Sprinkle contain spaces and perhaps other characters that are not allowed in domains. So we are going to write a program called Domainify; it converts a line of text into an acceptable domain segment and adds an appropriate **Top-level Domain (TLD)** to the end. Alongside the `sprinkle` folder, create a new one called `domainify` and add the `main.go` file with the following code:

```
package main
var tlds = []string{"com", "net"}
const allowedChars = "abcdefghijklmnopqrstuvwxyz0123456789_"
func main() {
    rand.Seed(time.Now().UTC().UnixNano())
    s := bufio.NewScanner(os.Stdin)
    for s.Scan() {
        text := strings.ToLower(s.Text())
        var newText []rune
        for _, r := range text {
            if unicode.IsSpace(r) {
                r = '-'
            }
            if !strings.ContainsRune(allowedChars, r) {
                continue
            }
            newText = append(newText, r)
        }
        fmt.Println(string(newText) + "." +
            tlds[rand.Intn(len(tlds))])
    }
}
```

You will notice a few similarities between Domainify and the Sprinkle program: we set the random seed using `rand.Seed`, generate a `NewScanner` method wrapping the `os.Stdin` reader, and scan each line until there is no more input.

We then convert the text to lowercase and build up a new slice of `rune` types called `newText`. The `rune` types consist of only characters that appear in the `allowedChars` string, which `strings.ContainsRune` lets us know. If `rune` is a space that we determine by calling `unicode.IsSpace`, we replace it with a hyphen, which is an acceptable practice in domain names.



Ranging over a string returns the index of each character and a `rune` type, which is a numerical value (specifically, `int32`) representing the character itself. For more information about runes, characters, and strings, refer to <http://blog.golang.org/strings>.

Finally, we convert `newText` from a `[]rune` slice into a string and add either `.com` or `.net` at the end, before printing it out using `fmt.Println`.

Let's build and run `Domainify`:

```
go build -o domainify  
./domainify
```

Type in some of these options to see how `domainify` reacts:

- Monkey
- Hello Domainify
- "What's up?"
- One (two) three!

You can see that, for example, `One (two) three!` might yield `one-two-three.com`.

We are now going to compose `Sprinkle` and `Domainify` to see them work together. In your terminal, navigate to the parent folder (probably `$GOPATH/src`) of `sprinkle` and `domainify` and run the following command:

```
./sprinkle/sprinkle | ./domainify/domainify
```

Here, we ran the `sprinkle` program and piped the output to the `domainify` program. By default, `sprinkle` uses the terminal as the input and `domainify` outputs to the terminal. Try typing in `chat` a few times again and notice the output is similar to what `Sprinkle` was outputting previously, except now they are acceptable for domain names. It is this piping between programs that allows us to compose command-line tools together.



Only supporting `.com` and `.net` top-level domains is fairly limiting. As an additional assignment, see whether you can accept a list of TLDs via a command-line flag.

Coolify

Often, domain names for common words, such as `chat`, are already taken, and a common solution is to play around with the vowels in the words. For example, we might remove a and make it `cht` (which is actually less likely to be available) or add a to produce `chaat`. While this clearly has no actual effect on coolness, it has become a popular, albeit slightly dated, way to secure domain names that still sound like the original word.

Our third program, Coolify, will allow us to play with the vowels of words that come in via the input and write modified versions to the output.

Create a new folder called `coolify` alongside `sprinkle` and `domainify`, and create the `main.go` code file with the following code:

```
package main
const (
    duplicateVowel bool    = true
    removeVowel   bool    = false
)
func randBool() bool {
    return rand.Intn(2) == 0
}
func main() {
    rand.Seed(time.Now().UTC().UnixNano())
    s := bufio.NewScanner(os.Stdin)
    for s.Scan() {
        word := []byte(s.Text())
        if randBool() {
            var vI int = -1
            for i, char := range word {
                switch char {
                case 'a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U':
                    if randBool() {
                        vI = i
                    }
                }
            }
            if vI >= 0 {
                switch randBool() {
                case duplicateVowel:
                    word = append(word[:vI+1], word[vI:]...)
                case removeVowel:
                    word = append(word[:vI], word[vI+1:]...)
                }
            }
        }
    }
}
```

```
    fmt.Println(string(word))
}
}
```

While the preceding Coolify code looks very similar to the code of Sprinkle and Domainify, it is slightly more complicated. At the very top of the code, we declare two constants, `duplicateVowel` and `removeVowel`, that help make the Coolify code more readable. The `switch` statement decides whether we duplicate or remove a vowel. Also, using these constants, we are able to express our intent very clearly, rather than use just `true` or `false`.

We then define the `randBool` helper function that just randomly returns either `true` or `false`. This is done by asking the `rand` package to generate a random number and confirming whether that number comes out as zero. It will be either `0` or `1`, so there's a fifty-fifty chance of it being `true`.

The `main` function of Coolify starts the same way as that of Sprinkle and Domainify setting the `rand.Seed` method and creating a scanner of the standard input stream before executing the loop body for each line of input. We call `randBool` first to decide whether we are even going to mutate a word or not, so Coolify will only affect half the words passed through it.

We then iterate over each rune in the string and look for a vowel. If our `randBool` method returns `true`, we keep the index of the vowel character in the `vI` variable. If not, we keep looking through the string for another vowel, which allows us to randomly select a vowel from the words rather than always modify the same one.

Once we have selected a vowel, we use `randBool` again to randomly decide what action to take.

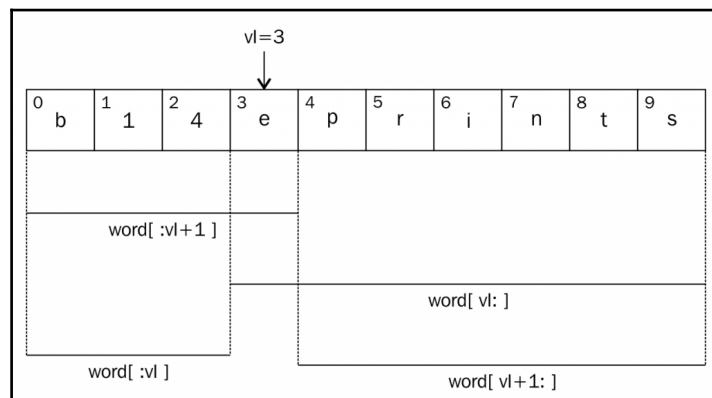


This is where the helpful constants come in; consider the following alternative switch statement:

```
switch randBool() { case true:  
    word = append(word[:vI+1], word[vI:]...) case false:  
    word = append(word[:vI], word[vI+1:]...) }
```

In the preceding code snippet, it's difficult to tell what is going on because `true` and `false` don't express any context. On the other hand, using `duplicateVowel` and `removeVowel` tells anyone reading the code what we mean by the result of `randBool`.

The three dots following the slices cause each item to pass as a separate argument to the `append` function. This is an idiomatic way of appending one slice to another. Inside the `switch` case, we do some slice manipulation to either duplicate the vowel or remove it altogether. We are slicing our `[]byte` slice again and using the `append` function to build a new one made up of sections of the original word. The following diagram shows which sections of the string we access in our code:



If we take the value `blueprints` as an example word and assume that our code has selected the first `e` character as the vowel (so that `vI` is 3), the following table will illustrate what each new slice of the word will represent:

Code	Value	Description
<code>word[:vI+1]</code>	blue	This describes the slice from the beginning of the word until the selected vowel. The <code>+1</code> is required because the value following the colon does not include the specified index; rather, it slices up to that value.
<code>word[vI :]</code>	eprints	This describes the slice starting from and including the selected vowel to the end of the slice.
<code>word[:vI]</code>	blu	This describes the slice from the beginning of the word up to, but not including, the selected vowel.
<code>word[vI+1 :]</code>	prints	This describes the slice from the item following the selected vowel to the end of the slice.

After we modify the word, we print it out using `fmt.Println`.

Let's build Coolify and play with it to see what it can do:

```
go build -o coolify  
./coolify
```

When Coolify is running, try typing `blueprints` to see what sort of modifications it comes up with:

```
blueprnts  
bleprnts  
bluepriints  
blueprnts  
blueprnts  
blueprnts  
bluprints
```

Let's see how Coolify plays with Sprinkle and Domainify by adding their names to our pipe chain. In the terminal, navigate back (using the `cd` command) to the parent folder and run the following commands:

```
./coolify/coolify | ./sprinkle/sprinkle | ./domainify/domainify
```

We will first spice up a word with extra pieces and make it cooler by tweaking the vowels before finally transforming it into a valid domain name. Play around by typing in a few words and seeing what suggestions our code makes.



Coolify only works on vowels; as an additional exercise, see whether you can make the code operate on every character it encounters just to see what happens.

Synonyms

So far, our programs have only modified words, but to really bring our solution to life, we need to be able to integrate a third-party API that provides word synonyms. This allows us to suggest different domain names while retaining the original meaning. Unlike Sprinkle and Domainify, Synonyms will write out more than one response for each word given to it. Our architecture of piping programs together means this won't be much of a problem; in fact, we do not even have to worry about it since each of the three programs is capable of reading multiple lines from the input source.

Big Huge Thesaurus, <http://bighugelabs.com/>, has a very clean and simple API that allows us to make a single HTTP GET request to look up synonyms.



In future, if the API we are using changes or disappears (after all, we're dealing with the Internet), you will find some options at <https://github.com/matryer/goblueprints>.

Before you can use Big Huge Thesaurus, you'll need an API key, which you can get by signing up to the service at <http://words.bighugelabs.com/>.

Using environment variables for configuration

Your API key is a sensitive piece of configuration information that you don't want to share with others. We could store it as `const` in our code. However, this would mean we will not be able to share our code without sharing our key (not good, especially if you love open source projects). Additionally, perhaps more importantly, you will have to recompile your entire project if the key expires or if you want to use a different one (you don't want to get into such a situation).

A better solution is using an environment variable to store the key, as this will allow you to easily change it if you need to. You could also have different keys for different deployments; perhaps you could have one key for development or testing and another for production. This way, you can set a specific key for a particular execution of code so you can easily switch between keys without having to change your system-level settings. Also, different operating systems deal with environment variables in similar ways, so they are a perfect choice if you are writing cross-platform code.

Create a new environment variable called `BHT_APIKEY` and set your API key as its value.



For machines running a bash shell, you can modify your `~/.bashrc` file or similar to include `export` commands, such as the following: `export BHT_APIKEY=abc123def456ghi789jkl` On Windows machines, you can navigate to the properties of your computer and look for **Environment Variables** in the **Advanced** section.

Consuming a web API

Making a request for <http://words.bighugelabs.com/apisample.php?v=2&format=json> in a web browser shows us what the structure of JSON response data looks like when finding synonyms for the word `love`:

```
{  
  "noun": {
```

```
"syn": [
    "passion",
    "beloved",
    "dear"
]
},
"verb": {
    "syn": [
        "love",
        "roll in the hay",
        "make out"
    ],
    "ant": [
        "hate"
    ]
}
}
```

A real API will return a lot more actual words than what is printed here, but the structure is the important thing. It represents an object, where the keys describe the types of word (verbs, nouns, and so on). Also, values are objects that contain arrays of strings keyed on syn or ant (for the synonym and antonym, respectively); it is the synonyms we are interested in.

To turn this JSON string data into something we can use in our code, we must decode it into structures of our own using the capabilities found in the `encoding/json` package. Because we're writing something that could be useful outside the scope of our project, we will consume the API in a reusable package rather than directly in our program code. Create a new folder called `thesaurus` alongside your other program folders (in `$GOPATH/src`) and insert the following code into a new `bighuge.go` file:

```
package thesaurus
import (
    "encoding/json"
    "errors"
    "net/http"
)
type BigHuge struct {
    APIKey string
}
type synonyms struct {
    Noun *words `json:"noun"`
    Verb *words `json:"verb"`
}
type words struct {
    Syn []string `json:"syn"`
}
```

```
}

func (b *BigHuge) Synonyms(term string) ([]string, error) {
    var syns []string
    response, err := http.Get("http://words.bighthugelabs.com/api/2/" + b.APIKey + "/" + term + "/json")
    if err != nil {
        return syns, errors.New("bighuge: Failed when looking for synonyms for " + term + " " + err.Error())
    }
    var data synonyms
    defer response.Body.Close()
    if err := json.NewDecoder(response.Body).Decode(&data); err != nil {
        return syns, err
    }
    if data.Noun != nil {
        syns = append(syns, data.Noun.Syn...)
    }
    if data.Verb != nil {
        syns = append(syns, data.Verb.Syn...)
    }
    return syns, nil
}
```

In the preceding code, the `BigHuge` type we define houses the necessary API key and provides the `Synonyms` method that will be responsible for doing the work of accessing the endpoint, parsing the response, and returning the results. The most interesting parts of this code are the `synonyms` and `words` structures. They describe the JSON response format in Go terms, namely an object containing noun and verb objects, which in turn contain a slice of strings in a variable called `Syn`. The tags (strings in backticks following each field definition) tell the `encoding/json` package which fields to map to which variables; this is required since we have given them different names.



Typically in JSON, keys have lowercase names, but we have to use capitalized names in our structures so that the `encoding/json` package would also know that the fields exist. If we don't, the package would simply ignore the fields. However, the types themselves (`synonyms` and `words`) do not need to be exported.

The `Synonyms` method takes a `term` argument and uses `http.Get` to make a web request to the API endpoint in which the URL contains not only the API key value, but also the `term` value itself. If the web request fails for some reason, we will make a call to `log.Fatalln`, which will write the error to the standard error stream and exit the program with a non-zero exit code (actually an exit code of 1). This indicates that an error has occurred.

If the web request is successful, we pass the response body (another `io.Reader`) to the `json.NewDecoder` method and ask it to decode the bytes into the `data` variable that is of our `synonyms` type. We defer the closing of the response body in order to keep the memory clean before using Go's built-in `append` function to concatenate both noun and verb synonyms to the `syns` slice that we then return.

Although we have implemented the `BigHuge` thesaurus, it isn't the only option out there, and we can express this by adding a `Thesaurus` interface to our package. In the `thesaurus` folder, create a new file called `thesaurus.go` and add the following interface definition to the file:

```
package thesaurus
type Thesaurus interface {
    Synonyms(term string) ([]string, error)
}
```

This simple interface just describes a method that takes a `term` string and returns either a slice of strings containing the synonyms or an error (if something goes wrong). Our `BigHuge` structure already implements this interface, but now, other users could add interchangeable implementations for other services, such as <http://www.dictionary.com/> or the Merriam-Webster online service.

Next, we are going to use this new package in a program. Change the directory in the terminal back up a level to `$GOPATH/src`, create a new folder called `synonyms`, and insert the following code into a new `main.go` file you will place in this folder:

```
func main() {
    apiKey := os.Getenv("BHT_APIKEY")
    thesaurus := &thesaurus.BigHuge{APIKey: apiKey}
    s := bufio.NewReader(os.Stdin)
    for s.Scan() {
        word := s.Text()
        syns, err := thesaurus.Synonyms(word)
        if err != nil {
            log.Fatalln("Failed when looking for synonyms for " + word + ", err")
        }
        if len(syns) == 0 {
            log.Fatalln("Couldn't find any synonyms for " + word + " ")
        }
        for _, syn := range syns {
            fmt.Println(syn)
        }
    }
}
```

Now when you manage your imports again, you will have written a complete program that is capable of looking up synonyms of words by integrating the Big Huge Thesaurus API.

In the preceding code, the first thing our `main` function does is that it gets the `BHT_APIKEY` environment variable value via the `os.Getenv` call. To protect your code, you might consider double-checking it to ensure the value is properly set; if not, report the error. For now, we will assume that everything is configured properly.

Next, the preceding code starts to look a little familiar since it scans each line of input again from `os.Stdin` and calls the `Synonyms` method to get a list of the replacement words.

Let's build a program and see what kind of synonyms the API comes back with when we input the word `chat`:

```
go build -o synonyms
./synonyms
chat
confab
confabulation
schmooze
New World chat
Old World chat
conversation
thrush
wood warbler
chew the fat
shoot the breeze
chitchat
chatter
```

The results you get will most likely differ from what we have listed here since we're hitting a live API. However, the important thing is that when we provide a word or term as an input to the program, it returns a list of synonyms as the output, one per line.

Getting domain suggestions

By composing the four programs we have built so far in this chapter, we already have a useful tool for suggesting domain names. All we have to do now is to run the programs while piping the output to the input in an appropriate way. In a terminal, navigate to the parent folder and run the following single line:

```
./synonyms/synonyms | ./sprinkle/sprinkle | ./coolify/coolify |
./domainify/domainify
```

Because the `synonyms` program is first in our list, it will receive the input from the terminal (whatever the user decides to type in). Similarly, because `domainify` is last in the chain, it will print its output to the terminal for the user to see. Along the way, the lines of words will be piped through other programs, giving each of them a chance to do their magic.

Type in a few words to see some domain suggestions; for example, when you type `chat` and hit return, you may see the following:

```
getcnfab.com
confabulationtim.com
getschmoozee.net
schmosee.com
neew-world-chatsite.net
oold-world-chatsite.com
conversatin.net
new-world-warblersit.com
gothrush.net
lets-wood-wrbler.com
chw-the-fat.com
```

The number of suggestions you get will actually depend on the number of synonyms. This is because it is the only program that generates more lines of output than what we input.

We still haven't solved our biggest problem: the fact that we have no idea whether the suggested domain names are actually available or not. So we still have to sit and type each one of them into a website. In the next section, we will address this issue.

Available

Our final program, `Available`, will connect to a WHOIS server to ask for details about the domains passed to it of course, if no details are returned, we can safely assume that the domain is available for purchase. Unfortunately, the WHOIS specification (see <http://tools.ietf.org/html/rfc3912>) is very small and contains no information about how a WHOIS server should reply when you ask for details about a domain. This means programmatically parsing the response becomes a messy endeavor. To address this issue for now, we will integrate with only a single WHOIS server, which we can be sure will have `No match` somewhere in the response when it has no records for the domain.



A more robust solution is to have a WHOIS interface with a well-defined structure for the details and perhaps an error message for cases when the domain doesn't exist with different implementations for different WHOIS servers. As you can imagine, it's quite a project; it is perfect for an open source effort.

Create a new folder called `available` alongside others and add a `main.go` file to it containing the following function code:

```
func exists(domain string) (bool, error) {
    const whoisServer string = "com.whois-servers.net"
    conn, err := net.Dial("tcp", whoisServer+":43")
    if err != nil {
        return false, err
    }
    defer conn.Close()
    conn.Write([]byte(domain + "\r\n"))
    scanner := bufio.NewScanner(conn)
    for scanner.Scan() {
        if strings.Contains(strings.ToLower(scanner.Text()), "no match") {
            return false, nil
        }
    }
    return true, nil
}
```

The `exists` function implements what little there is in the WHOIS specification by opening a connection to port 43 on the specified `whoisServer` instance with a call to `net.Dial`. We then defer the closing of the connection, which means that no matter how the function exits (successful, with an error, or even a panic), `Close()` will still be called on the `conn` connection. Once the connection is open, we simply write the domain followed by `\r\n` (the carriage return and linefeed characters). This is all that the specification tells us, so we are on our own from now on.

Essentially, we are looking for some mention of "no match" in the response, and this is how we will decide whether a domain exists or not (`exists` in this case is actually just asking the WHOIS server whether it has a record for the domain we specified). We use our favorite `bufio.Scanner` method to help us iterate over the lines in the response. Passing the connection to `NewScanner` works because `net.Conn` is actually an `io.Reader` too. We use `strings.ToLower` so we don't have to worry about case sensitivity and `strings.Contains` to check whether any one of the lines contains the `no match` text. If it does, we return `false` (since the domain doesn't exist); otherwise, we return `true`.

The `com.whois-servers.net` WHOIS service supports domain names for `.com` and `.net`, which is why the `Domainify` program only adds these types of domains. If you had used a server that had WHOIS information for a wider selection of domains, you could have added support for additional TLDs.

Let's add a `main` function that uses our `exists` function to check whether the incoming domains are available or not. The check mark and cross mark symbols in the following code are optional if your terminal doesn't support them you are free to substitute them with simple `Yes` and `No` strings.

Add the following code to `main.go`:

```
var marks = map[bool]string{true: "✓", false: "✗"}  
func main() {  
    s := bufio.NewReader(os.Stdin)  
    for s.Scan() {  
        domain := s.Text()  
        fmt.Print(domain, " ")  
        exist, err := exists(domain)  
        if err != nil {  
            log.Fatalln(err)  
        }  
        fmt.Println(marks[!exist])  
        time.Sleep(1 * time.Second)  
    }  
}
```



We can use the check and cross characters in our code happily because all Go code files are UTF-8 compliant the best way to actually get these characters is to search the Web for them and use the copy and paste option to bring them into our code. Otherwise, there are platform-dependent ways to get such special characters.

In the preceding code for the `main` function, we simply iterate over each line coming in via `os.Stdin`. This process helps us print out the domain with `fmt.Print` (but not `fmt.Println`, as we do not want the linefeed yet), call our `exists` function to check whether the domain exists or not, and print out the result with `fmt.Println` (because we *do* want a linefeed at the end).

Finally, we use `time.Sleep` to tell the process to do nothing for a second in order to make sure we take it easy on the WHOIS server.



Most WHOIS servers will be limited in various ways in order to prevent you from taking up too much in terms of resources. So, slowing things down is a sensible way to make sure we don't make the remote servers angry. Consider what this also means for unit tests. If a unit test were actually making real requests to a remote WHOIS server, every time your tests run, you will be clocking up statistics against your IP address. A much better approach would be to stub the WHOIS server to simulate responses.

The `marks` map at the top is a nice way to map the `bool` response from `exists` to human-readable text, allowing us to just print the response in a single line using `fmt.Println(marks[!exist])`. We are saying *not exist* because our program is checking whether the domain is available or not (logically, the opposite of whether it exists in the WHOIS server or not).

After fixing the import statements for the `main.go` file, we can try out `Available` to see whether the domain names are available or not by typing the following command:

```
go build -o available
./available
```

Once `Available` is running, type in some domain names and see the result appear on the next line:

```
packtpub.com
packtpub.com ✗
matryer.com
matryer.com ✗
made-up-domain-12345678.net
made-up-domain-12345678.net ✓
```

As you can see, for domains that are not available, we get a little cross mark next to them; however, when we make up a domain name using random numbers, we see that it is indeed available.

Composing all five programs

Now that we have completed all five programs, it's time to put them all together so that we can use our tool to find an available domain name for our chat application. The simplest way to do this is to use the technique we have been using throughout this chapter: using pipes in a terminal to connect the output and input.

In the terminal, navigate to the parent folder of the five programs and run the following single line of code:

```
./synonyms/synonyms | ./sprinkle/sprinkle | ./coolify/coolify |  
./domainify/domainify | ./available/available
```

Once the programs are running, type in a starting word and see how it generates suggestions before checking their availability.

For example, typing in `chat` might cause the programs to take the following actions:

1. The word `chat` goes into `synonyms`, which results in a series of synonyms:
 - `confab`
 - `confabulation`
 - `schmooze`
2. The synonyms flow into `sprinkle`; here they are augmented with web-friendly prefixes and suffixes, such as the following:
 - `confabapp`
 - `goconfabulation`
 - `schmooze time`
3. These new words flow into `coolify`; here the vowels are potentially tweaked:
 - `confabaapp`
 - `goconfabulatioon`
 - `schmoooze time`

4. The modified words then flow into `domainify`; here they are turned into valid domain names:
 - confabaapp.com
 - goconfabulatioon.net
 - schmooze-time.com
5. Finally, the domain names flow into `available`; here they are checked against the WHOIS server to see whether somebody has already taken the domain or not:
 - confabaapp.com X
 - goconfabulatioon.net ✓
 - schmooze-time.com ✓

One program to rule them all

Running our solution by piping programs together is an elegant form of architecture, but it doesn't have a very elegant interface. Specifically, whenever we want to run our solution, we have to type the long, messy line where each program is listed and separated by pipe characters. In this section, we are going to write a Go program that uses the `os/exec` package to run each subprogram while piping the output from one to the input of the next, as per our design.

Create a new folder called `domainfinder` alongside the other five programs and create another new folder called `lib` inside this folder. The `lib` folder is where we will keep builds of our subprograms, but we don't want to copy and paste them every time we make a change. Instead, we will write a script that builds the subprograms and copies the binaries to the `lib` folder for us.

Create a new file called `build.sh` on Unix machines or `build.bat` for Windows and insert into it the following code:

```
#!/bin/bash
echo Building domainfinder...
go build -o domainfinder
echo Building synonyms...
cd ../synonyms
go build -o ../domainfinder/lib/synonyms
echo Building available...
cd ../available
go build -o ../domainfinder/lib/available
```

```
cd ../build
echo Building sprinkle...
cd ../sprinkle
go build -o ../domainfinder/lib/sprinkle
cd ../build
echo Building coolify...
cd ../coolify
go build -o ../domainfinder/lib/coolify
cd ../build
echo Building domainify...
cd ../domainify
go build -o ../domainfinder/lib/domainify
cd ../build
echo Done.
```

The preceding script simply builds all our subprograms (including `domainfinder`, which we are yet to write), telling `go build` to place them in our `lib` folder. Be sure to give execution rights to the new script by doing `chmod +x build.sh` or something similar. Run this script from a terminal and look inside the `lib` folder to ensure that it has indeed placed the binaries for our subprograms.



Don't worry about the `no buildable Go source files` error for now; it's just Go telling us that the `domainfinder` program doesn't have any `.go` files to build.

Create a new file called `main.go` inside `domainfinder` and insert the following code into the file:

```
package main
var cmdChain = []*exec.Cmd{
    exec.Command("lib/synonyms"),
    exec.Command("lib/sprinkle"),
    exec.Command("lib/coolify"),
    exec.Command("lib/domainify"),
    exec.Command("lib/available"),
}
func main() {
    cmdChain[0].Stdin = os.Stdin
    cmdChain[len(cmdChain)-1].Stdout = os.Stdout
    for i := 0; i < len(cmdChain)-1; i++ {
        thisCmd := cmdChain[i]
        nextCmd := cmdChain[i+1]
        stdout, err := thisCmd.StdoutPipe()
        if err != nil {
            log.Fatalln(err)
        }
    }
}
```

```
    nextCmd.Stdin = stdout
}
for _, cmd := range cmdChain {
    if err := cmd.Start(); err != nil {
        log.Fatalln(err)
    } else {
        defer cmd.Process.Kill()
    }
}
for _, cmd := range cmdChain {
    if err := cmd.Wait(); err != nil {
        log.Fatalln(err)
    }
}
}
```

The `os/exec` package gives us everything we need to work with to run external programs or commands from within Go programs. First, our `cmdChain` slice contains `*exec.Cmd` commands in the order in which we want to join them together.

At the top of the `main` function, we tie the `Stdin` (standard in stream) of the first program with the `os.Stdin` stream of this program and the `Stdout` (standard out stream) of the last program with the `os.Stdout` stream of this program. This means that, like before, we will be taking input through the standard input stream and writing output to the standard output stream.

Our next block of code is where we join the subprograms together by iterating over each item and setting its `Stdin` to the `Stdout` stream of the program before it.

The following table shows each program with a description of where it gets its input from and where its output goes:

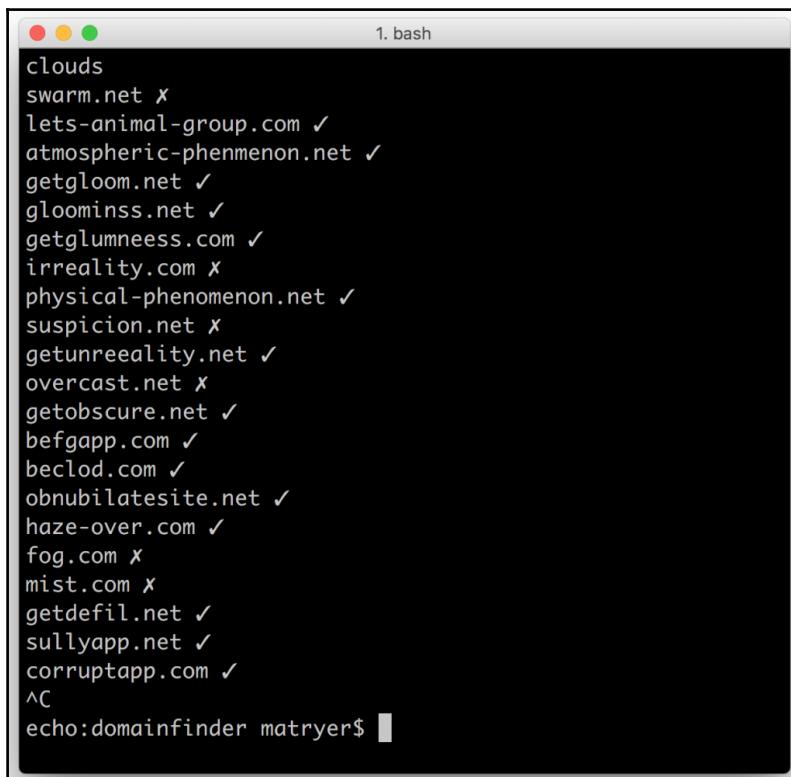
Program	Input (Stdin)	Output (Stdout)
synonyms	The same Stdin as domainfinder	sprinkle
sprinkle	synonyms	coolify
coolify	sprinkle	domainify
domainify	coolify	available
available	domainify	The same Stdout as domainfinder

We then iterate over each command calling the `Start` method, which runs the program in the background (as opposed to the `Run` method, which will block our code until the subprogram exists which would be no good since we will have to run five programs at the same time). If anything goes wrong, we bail with `log.FatalIn`; however, if the program starts successfully, we defer a call to kill the process. This helps us ensure the subprograms exit when our `main` function exits, which will be when the `domainfinder` program ends.

Once all the programs start running, we iterate over every command again and wait for it to finish. This is to ensure that `domainfinder` doesn't exit early and kill off all the subprograms too soon.

Run the `build.sh` or `build.bat` script again and notice that the `domainfinder` program has the same behavior as we have seen before, with a much more elegant interface.

The following screenshot shows the output from our programs when we type `clouds`; we have found quite a few available domain name options:



```
1. bash
clouds
swarm.net ✘
lets-animal-group.com ✓
atmospheric-phenmenon.net ✓
getgloom.net ✓
gloominss.net ✓
getglumneess.com ✓
irreality.com ✘
physical-phenomenon.net ✓
suspicion.net ✘
getunreality.net ✓
overcast.net ✘
getobscure.net ✓
befgapp.com ✓
beclod.com ✓
obnubilatesite.net ✓
haze-over.com ✓
fog.com ✘
mist.com ✘
getdefil.net ✓
sullyapp.net ✓
corruptapp.com ✓
^C
echo:domainfinder matryer$
```

Summary

In this chapter, we learned how five small command-line programs can, when composed together, produce powerful results while remaining modular. We avoided tightly coupling our programs so they could still be useful in their own right. For example, we can use our Available program just to check whether the domain names we manually enter are available or not, or we can use our `synonyms` program just as a command-line thesaurus.

We learned how standard streams could be used to build different flows of these types of programs and how the redirection of standard input and standard output lets us play around with different flows very easily.

We learned how simple it is in Go to consume a JSON RESTful API web service when we wanted to get the synonyms from Big Huge Thesaurus. We also consumed a non-HTTP API when we opened a connection to the WHOIS server and wrote data over raw TCP.

We saw how the `math/rand` package can bring a little variety and unpredictability by allowing us to use pseudo random numbers and decisions in our code, which means that each time we run our program, we will get different results.

Finally, we built our `domainfinder` super program that composes all the subprograms together, giving our solution a simple, clean, and elegant interface.

In the next chapter, we will take some ideas we have learned so far one step further by exploring how to connect programs using messaging queue technologies allowing them to be distributed across many machines to achieve large scale.

5

Building Distributed Systems and Working with Flexible Data

In this chapter, we will explore transferrable skills that allow us to use schemaless data and distributed technologies to solve big data problems. The system we will build in this chapter will prepare us for a future where all democratic elections happen online on Twitter, of course. Our solution will collect and count votes by querying Twitter's streaming API for mentions of specific hash tags, and each component will be capable of horizontally scaling to meet demand. Our use case is a fun and interesting one, but the core concepts we'll learn and the specific technology choices we'll make are the real focus of this chapter. The ideas discussed here are directly applicable to any system that needs true-scale capabilities.



Horizontal scaling refers to adding nodes, such as physical machines, to a system in order to improve its availability, performance, and/or capacity. Big data companies such as Google can scale by adding affordable and easy-to-obtain hardware (commonly referred to as commodity hardware) due to the way they write their software and architect their solutions.

Vertical scaling is synonymous to increasing the resource available to a single node, such as adding additional RAM to a box or a processor with more cores.

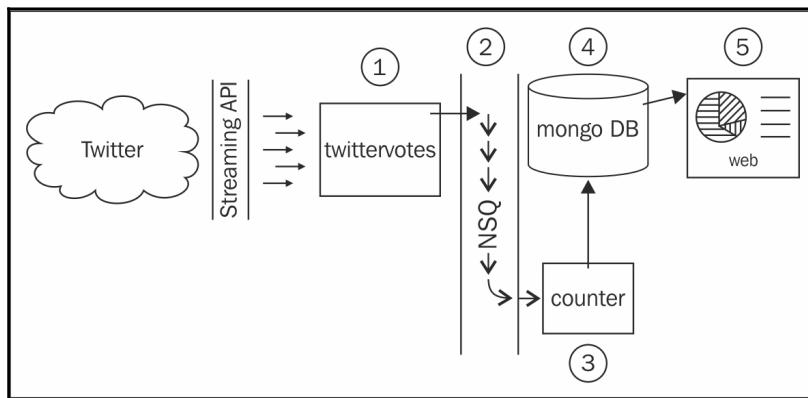
In this chapter, you will:

- Learn about distributed **NoSQL** datastores, specifically how to interact with MongoDB
- Learn about **distributed messaging queues**, in our case, Bit.ly's NSQ and how to use the `go-nsq` package to easily publish and subscribe to events

- Stream live tweet data through Twitter's streaming APIs and manage long running net connections
- Learn how to properly stop programs with many internal goroutines
- Learn how to use low memory channels for signaling

The system design

Having a basic design sketched out is often useful, especially in distributed systems where many components will be communicating with each other in different ways. We don't want to spend too long on this stage because our design is likely to evolve as we get stuck into the details, but we will look at a high-level outline so that we can discuss the constituents and how they fit together:



The preceding diagram shows the basic overview of the system we are going to build:

- Twitter is the social media network we all know and love.
- Twitter's streaming API allows long-running connections where tweet data is streamed as quickly as possible.
- `twittervotes` is a program we will write that pulls the relevant tweet data via the Twitter API, decides what is being voted for (rather, which options are mentioned in the tweet body), and then pushes the vote into NSQ.
- NSQ is an open source, real-time distributed messaging platform designed to operate at scale, built and maintained by Bit.ly. NSQ carries the message across its instances, making it available to anyone who has expressed an interest in the vote data.

- `counter` is a program we will write that listens out for votes on the messaging queue and periodically saves the results in the MongoDB database. It receives the vote messages from NSQ and keeps an in-memory tally of the results, periodically pushing an update to persist the data.
- MongoDB is an open source document database designed to operate at scale.
- `web` is a web server program that will expose the live results that we will write in the next chapter.

It could be argued that a single Go program could be written that reads the tweets, counts the votes, and pushes them to a user interface, but such a solution, while being a great proof of concept, would be very limited in scale. In our design, any one of the components can be horizontally scaled as the demand for that particular capability increases. If we have relatively few polls but lots of people viewing the data, we can keep the `twittervotes` and `counter` instances down and add more `web` and MongoDB nodes or vice versa if the situation is reversed.



Another key advantage to our design is redundancy; since we can have many instances of our components working at the same time, if one of our boxes disappears (due to a system crash or a power cut, for example), the others can pick up the slack. Modern architectures often distribute such a system over the geographical expanse in order to protect from local natural disasters too. All of these options are available for use if we build our solution in this way.

We chose specific technologies in this chapter because of their links to Go (NSQ, for example, is written entirely in Go) and the availability of well-tested drivers and packages. Conceptually, however, you can drop in a variety of alternatives as you see fit.

The database design

We will call our MongoDB database `ballots`. It will contain a single collection called `polls`, which is where we will store the poll details, such as the title, the options, and the results (in a single JSON document). The code for a poll will look something like this:

```
{  
  "_id": "???",  
  "title": "Poll title",  
  "options": ["one", "two", "three"],  
  "results": {  
    "one": 100,  
    "two": 200,  
    "three": 300
```

```
}
```

The `_id` field is a unique string for each item that is automatically generated by MongoDB. The `options` field contains an array of string options; these are the hash tags we will look for on Twitter. The `results` field is a map where the key represents the option, and the value represents the total number of votes for each item.

Installing the environment

The code we write in this chapter has real external dependencies that we need to set up before we can start to build our system.



Be sure to check out the chapter notes at <https://github.com/matryer/goblueprints> if you get stuck on installing any of the dependencies.

In most cases, services such as `mongod` and `nsqd` will have to be started before we can run our programs. Since we are writing components of a distributed system, we will have to run each program at the same time, which is as simple as opening many terminal windows.

Introducing NSQ

NSQ is a messaging queue that allows one program to send messages or events to another or to many other programs running either locally on the same machine or on different nodes connected by a network. NSQ guarantees the delivery of each message at least once, which means that it keeps undelivered messages cached until all interested parties have received them. This means that even if we stop our `counter` program, we won't miss any votes. You can contrast this capability with fire-and-forget message queues, where information is deemed out of date, and is, therefore, forgotten if it isn't delivered in time and when the sender of the messages doesn't care whether the consumer received them or not.

A message queue abstraction allows you to have different components of a system running in different places, provided that they have network connectivity to the queue. Your programs are decoupled from others; instead, your designs start to care about the ins and outs of specialized micro services rather than flow of data through a monolithic program.

NSQ transfers raw bytes, which means that it is up to us how we encode data into these bytes. For example, we could encode the data as JSON or in a binary format depending on our needs. In our case, we are going to send the vote option as a string without any additional encoding, since we are only sharing a single data field.

We first need to get NSQ installed and running:

1. Open <http://nsq.io/deployment/installing.html> in a browser (or search `install nsq`) and follow the instructions for your environment. You can either download precompiled binaries or build your own from the source. If you have homebrew installed, installing NSQ is as simple as typing the following:

```
brew install nsq
```

2. Once you have installed NSQ, you will need to add the `bin` folder to your `PATH` environment variable so that the tools are available in a terminal.
3. To validate that NSQ is properly installed, open a terminal and run `nsqlookupd`; if the program successfully starts, you should see output similar to the following:

```
nsqlookupd v0.2.27 (built w/go1.3)
TCP: listening on [::]:4160
HTTP: listening on [::]:4161
```

We are going to use the default ports to interact with NSQ, so take note of the TCP and HTTP ports listed in the output, as we will be referring to them in our code.

4. Press `Ctrl + C` to stop the process for now; we'll start them properly later.

The key tools from the NSQ installation that we are going to use are `nsqlookupd` and `nsqd`. The `nsqlookupd` program is a daemon that manages topology information about the distributed NSQ environment; it keeps track of all the `nsqd` producers for specific topics and provides interfaces for clients to query such information. The `nsqd` program is a daemon that does the heavy lifting for NSQ, such as receiving, queuing, and delivering messages from and to interested parties.

For more information and background on NSQ, visit <http://nsq.io/>.



NSQ driver for Go

The NSQ tools themselves are written in Go, so it is logical that the Bit.ly team already has a Go package that makes interacting with NSQ very easy. We will need to use it, so in a terminal, you can get it using `go get`:

```
go get github.com/bitly/go-nsq
```

Introducing MongoDB

MongoDB is a document database, which allows you to store and query JSON documents and the data within them. Each document goes into a collection that can be used to group the documents together without enforcing any schema on the data inside them. Unlike rows in a traditional RDBMS, such as Oracle, Microsoft SQL Server, or MySQL, it is perfectly acceptable for documents to have a different shape. For example, a `people` collection can contain the following three JSON documents at the same time:

```
{"name": "Mat", "lang": "en", "points": 57}  
{"name": "Laurie", "position": "Scrum Master"}  
{"position": "Traditional Manager", "exists": false}
```

This flexibility allows data with varying structures to coexist without impacting performance or wasting space. It is also extremely useful if you expect your software to evolve over time, as we really always should.

MongoDB was designed to scale while also remaining very easy to work with on single-box installations, such as our development machine. When we host our application for production, we would most likely install a more complex multi-sharded, replicated system, which is distributed across many nodes and locations, but for now, just running `mongod` will do.

Head over to <http://www.mongodb.org/downloads> in order to grab the latest version of MongoDB and install it, making sure to register the `bin` folder with your `PATH` environment variable, as usual.

To validate that MongoDB is successfully installed, run the `mongod` command, and then hit `Ctrl + C` to stop it for now.

MongoDB driver for Go

Gustavo Niemeyer has done a great job in simplifying interactions with MongoDB with his `mgo` (pronounced *mango*) package hosted at <http://labix.org/mgo>, which is *go gettable* with the following command:

```
go get gopkg.in/mgo.v2
```

Starting the environment

Now that we have all the pieces we need installed, we need to start our environment. In this section, we are going to:

- Start `nsqlookupd` so that our `nsqd` instances are discoverable
- Start `nsqd` and tell it which `nsqlookupd` to use
- Start `mongod` for data services

Each of these daemons should run in their own terminal window, which will make it easy for us to stop them by just hitting *Ctrl + C*.



Remember the page number for this section as you are likely to revisit it a few times as you work through this chapter.

In a terminal window, run the following:

```
nsqlookupd
```

Take note of the TCP port, which is 4160 by default, and in another terminal window, run the following:

```
nsqd --lookupd-tcp-address=localhost:4160
```

Make sure the port number in the `--lookupd-tcp-address` flag matches the TCP port of the `nsqlookupd` instance. Once you start `nsqd`, you will notice some output printed to the terminal from both `nsqlookupd` and `nsqd`; this indicates that the two processes are talking to each other.

In yet another window or tab, start MongoDB by running the following command:

```
mongod --dbpath ./db
```

The `dbpath` flag tells MongoDB where to store the data files for our database. You can pick any location you like, but you'll have to make sure the folder exists before `mongod` will run.



By deleting the `dbpath` folder at any time, you can effectively erase all data and start afresh. This is especially useful during development.

Now that our environment is running, we are ready to start building our components.

Reading votes from Twitter

In your `$GOPATH/src` folder, alongside other projects, create a new folder called `socialpoll` for this chapter. This folder won't be a Go package or a program by itself, but it will contain our three component programs. Inside `socialpoll`, create a new folder called `twittervotes` and add the obligatory `main.go` template (this is important as `main` packages without a `main` function won't compile):

```
package main
func main() {}
```

Our `twittervotes` program is going to:

- Load all polls from the MongoDB database using `mgo` and collect all options from the `options` array in each document
- Open and maintain a connection to Twitter's streaming APIs looking for any mention of the options
- Figure out which option is mentioned and push that option through to NSQ for each tweet that matches the filter
- If the connection to Twitter is dropped (which is common in long-running connections that are actually part of Twitter's streaming API specification) after a short delay (so that we do not bombard Twitter with connection requests), reconnect and continue
- Periodically re-query MongoDB for the latest polls and refresh the connection to Twitter to make sure we are always looking out for the right options
- Gracefully stop itself when the user terminates the program by hitting `Ctrl + C`

Authorization with Twitter

In order to use the streaming API, we will need authentication credentials from Twitter's Application Management console, much in the same way we did for our Gomniauth service providers in Chapter 3, *Three Ways to Implement Profile Pictures*. Head over to <https://apps.twitter.com> and create a new app called something like SocialPoll (the names have to be unique, so you can have some fun here; the choice of name doesn't affect the code either way). When your app has been created, visit the **API Keys** tab and locate the **Your access token** section, where you need to create a new access token. After a short delay, refresh the page and note that you, in fact, have two sets of keys and secrets: an API key and a secret and an access token and the corresponding secret. Following good coding practices, we are going to set these values as environment variables so that our program can have access to them without us having to hardcode them in our source files. The keys we will use in this chapter are as follows:

- SP_TWITTER_KEY
- SP_TWITTER_SECRET
- SP_TWITTER_ACESSTOKEN
- SP_TWITTER_ACESSECRET

You may set the environment variables however you like, but since the app relies on them in order to work, creating a new file called `setup.sh` (for bash shells) or `setup.bat` (on Windows) is a good idea since you can check such files into your source code repository. Insert the following code in `setup.sh` by copying the appropriate values from the Twitter app page:

```
#!/bin/bash
export SP_TWITTER_KEY=yC2EDnaNrEhN5fd33g...
export SP_TWITTER_SECRET=6n0rToIpskCo1ob...
export SP_TWITTER_ACESSTOKEN=2427-13677...
export SP_TWITTER_ACESSECRET=SpnZf336u...
```

On Windows, the code will look something like this:

```
SET SP_TWITTER_KEY=yC2EDnaNrEhN5fd33g...
SET SP_TWITTER_SECRET=6n0rToIpskCo1ob...
SET SP_TWITTER_ACESSTOKEN=2427-13677...
SET SP_TWITTER_ACESSECRET=SpnZf336u...
```

Run the file with the source or call commands to have the values set appropriately, or add them to your `.bashrc` or `C:\cmdauto.cmd` files to save you from running them every time you open a new terminal window.

If you're not sure how to do this, just search for `Setting environment variables on Linux` or something similar, and the Internet will help you.

Extracting the connection

The Twitter streaming API supports HTTP connections that stay open for a long time, and given the design of our solution, we are going to need to access the `net.Conn` object in order to close it from outside of the goroutine in which requests occur. We can achieve this by providing our own `dial` method to an `http.Transport` object that we will create.

Create a new file called `twitter.go` inside `twittersvotes` (which is where all things Twitter-related will live), and insert the following code:

```
var conn net.Conn
func dial(netw, addr string) (net.Conn, error) {
    if conn != nil {
        conn.Close()
        conn = nil
    }
    netc, err := net.DialTimeout(netw, addr, 5*time.Second)
    if err != nil {
        return nil, err
    }
    conn = netc
    return netc, nil
}
```

Our bespoke `dial` function first ensures that `conn` is closed and then opens a new connection, keeping the `conn` variable updated with the current connection. If a connection dies (Twitter's API will do this from time to time) or is closed by us, we can redial without worrying about zombie connections.

We will periodically close the connection ourselves and initiate a new one because we want to reload the options from the database at regular intervals. To do this, we need a function that closes the connection and also closes `io.ReadCloser`, which we will use to read the body of the responses. Add the following code to `twitter.go`:

```
var reader io.ReadCloser
func closeConn() {
    if conn != nil {
```

```
    conn.Close()
}
if reader != nil {
    reader.Close()
}
}
```

Now, we can call `closeConn` at any time in order to break the ongoing connection with Twitter and tidy things up. In most cases, our code will load the options from the database again and open a new connection right away, but if we're shutting the program down (in response to a *Ctrl + C* hit), then we can call `closeConn` just before we exit.

Reading environment variables

Next, we are going to write a function that will read the environment variables and set up the OAuth objects we'll need in order to authenticate the requests. Add the following code to the `twitter.go` file:

```
var (
    authClient *oauth.Client
    creds *oauth.Credentials
)
func setupTwitterAuth() {
    var ts struct {
        ConsumerKey     string `env:"SP_TWITTER_KEY,required"`
        ConsumerSecret string `env:"SP_TWITTER_SECRET,required"`
        AccessToken     string `env:"SP_TWITTER_ACCESTOKEN,required"`
        AccessSecret    string `env:"SP_TWITTER_ACCESSSECRET,required"`
    }
    if err := envdecode.Decode(&ts); err != nil {
        log.Fatalln(err)
    }
    creds = &oauth.Credentials{
        Token:  ts.AccessToken,
        Secret: ts.AccessSecret,
    }
    authClient = &oauth.Client{
        Credentials: oauth.Credentials{
            Token:  ts.ConsumerKey,
            Secret: ts.ConsumerSecret,
        },
    }
}
```

Here, we define a `struct` type to store the environment variables that we need to authenticate with Twitter. Since we don't need to use the type elsewhere, we define it inline and create a variable called `ts` of this anonymous type (that's why we have the somewhat unusual `var ts struct...` code). We then use Joe Shaw's `envdecode` package to pull in these environment variables for us. You will need to run `go get github.com/joeshaw/envdecode` and also import the `log` package. Our program will try to load appropriate values for all the fields marked `required` and return an error if it fails to do so, which reminds people that the program won't work without Twitter credentials.

The strings inside the back ticks alongside each field in `struct` are called tags and are available through a reflection interface, which is how `envdecode` knows which variables to look for. We added the `required` argument to this package, which indicates that it is an error for any of the environment variables to be missing (or empty).

Once we have the keys, we use them to create `oauth.Credentials` and an `oauth.Client` object from Gary Burd's `go-oauth` package, which will allow us to authorize requests with Twitter.

Now that we have the ability to control the underlying connection and authorize requests, we are ready to write the code that will actually build the authorized request and return the response. In `twitter.go`, add the following code:

```
var (
    authSetupOnce sync.Once
    httpClient    *http.Client
)
func makeRequest(req *http.Request, params url.Values) (*http.Response,
error) {
    authSetupOnce.Do(func() {
        setupTwitterAuth()
        httpClient = &http.Client{
            Transport: &http.Transport{
                Dial: dial,
            },
        }
    })
    formEnc := params.Encode()
    req.Header.Set("Content-Type", "application/x-www-form-urlencoded")
    req.Header.Set("Content-Length", strconv.Itoa(len(formEnc)))
    req.Header.Set("Authorization", authClient.AuthorizationHeader(creds,
        "POST",
        req.URL, params))
    return httpClient.Do(req)
}
```

We use `sync.Once` to ensure our initialization code gets run only once despite the number of times we call `makeRequest`. After calling the `setupTwitterAuth` method, we create a new `http.Client` function using an `http.Transport` function that uses our custom `dial` method. We then set the appropriate headers required for authorization with Twitter by encoding the specified `params` object that will contain the options we are querying for.

Reading from MongoDB

In order to load the polls, and therefore the options to search Twitter for, we need to connect to and query MongoDB. In `main.go`, add the two functions `dialdb` and `closedb`:

```
var db *mgo.Session
func dialdb() error {
    var err error
    log.Println("dialing mongodb: localhost")
    db, err = mgo.Dial("localhost")
    return err
}
func closedb() {
    db.Close()
    log.Println("closed database connection")
}
```

These two functions will connect to and disconnect from the locally running MongoDB instance using the `mgo` package and store `mgo.Session` (the database connection object) in a global variable called `db`.



As an additional assignment, see whether you can find an elegant way to make the location of the MongoDB instance configurable so that you don't need to run it locally.

Assuming MongoDB is running and our code is able to connect, we need to load the poll objects and extract all the options from the documents, which we will then use to search Twitter. Add the following `loadOptions` function to `main.go`:

```
type poll struct {
    Options []string
}
func loadOptions() ([]string, error) {
    var options []string
    iter := db.DB("ballots").C("polls").Find(nil).Iter()
    var p poll
    for iter.Next(&p) {
        options = append(options, p.Options)
    }
    return options, iter.Err()
}
```

```
    options = append(options, p.Options...)
}
iter.Close()
return options, iter.Err()
}
```

Our poll document contains more than just `Options`, but our program doesn't care about anything else, so there's no need for us to bloat our `poll` struct. We use the `db` variable to access the `polls` collection from the `ballots` database and call the `mgo` package's fluent `Find` method, passing `nil` (meaning no filtering).



A fluent interface (first coined by Eric Evans and Martin Fowler) refers to an API design that aims to make the code more readable by allowing you to chain method calls together. This is achieved by each method returning the context object itself so that another method can be called directly afterwards. For example, `mgo` allows you to write queries such as this:

```
query := col.Find(q).Sort("field").Limit(10).Skip(10)
```

We then get an iterator by calling the `Iter` method, which allows us to access each poll one by one. This is a very memory-efficient way of reading the poll data because it only ever uses a single `poll` object. If we were to use the `All` method instead, the amount of memory we'd use would depend on the number of polls we had in our database, which could be out of our control.

When we have a poll, we use the `append` method to build up the `options` slice. Of course, with millions of polls in the database, this slice too would grow large and unwieldy. For that kind of scale, we would probably run multiple `twittervotes` programs, each dedicated to a portion of the poll data. A simple way to do this would be to break polls into groups based on the letters the titles begin with, such as group A-N and O-Z. A somewhat more sophisticated approach would be to add a field to the `poll` document, grouping it up in a more controlled manner, perhaps based on the stats for the other groups so that we are able to balance the load across many `twittervotes` instances.



The `append` built-in function is actually a variadic function, which means you can pass multiple elements for it to append. If you have a slice of the correct type, you can add `...` to the end, which simulates the passing of each item of the slice as a different argument.

Finally, we close the iterator and clean up any used memory before returning the options and any errors that occurred while iterating (by calling the `Err` method in the `mgo.Iter` object).

Reading from Twitter

Now we are able to load the options and make authorized requests to the Twitter API. We are ready to write the code that initiates the connection and continuously reads from the stream until either we call our `closeConn` method or Twitter closes the connection for one reason or another. The structure contained in the stream is a complex one, containing all kinds of information about the tweet who made it and when and even what links or mentions of users occur in the body (refer to Twitter's API documentation for more details). However, we are only interested in the tweet text itself; so, don't worry about all the other noise and add the following structure to `twitter.go`:

```
type tweet struct {
    Text string
}
```



This may feel incomplete, but think about how clear it makes our intentions to other programmers who might see our code: a tweet has some text, and that is all we care about.

Using this new structure, in `twitter.go`, add the following `readFromTwitter` function that takes a send only channel called `votes`; this is how this function will inform the rest of our program that it has noticed a vote on Twitter:

```
func readFromTwitter(votes chan<- string) {
    options, err := loadOptions()
    if err != nil {
        log.Println("failed to load options:", err)
        return
    }
    u, err := url.Parse("https://stream.twitter.com/1.1/statuses
/filter.json")
    if err != nil {
        log.Println("creating filter request failed:", err)
        return
    }
    query := make(url.Values)
    query.Set("track", strings.Join(options, ","))
    req, err := http.NewRequest("POST", u.String(), strings.NewReader
(query.Encode()))
    if err != nil {
        log.Println("creating filter request failed:", err)
        return
    }
    resp, err := makeRequest(req, query)
    if err != nil {
```

```
    log.Println("making request failed:", err)
    return
}
reader := resp.Body
decoder := json.NewDecoder(reader)
for {
    var t tweet
    if err := decoder.Decode(&t); err != nil {
        break
    }
    for _, option := range options {
        if strings.Contains(
            strings.ToLower(t.Text),
            strings.ToLower(option),
        ) {
            log.Println("vote:", option)
            votes <- option
        }
    }
}
}
```

In the preceding code, after loading the options from all the polls data (by calling the `loadOptions` function), we use `url.Parse` to create a `url.URL` object that describes the appropriate endpoint on Twitter. We build a `url.Values` object called `query` and set the options as a comma-separated list. As per the API, we make a new `POST` request using the encoded `url.Values` object as the body and pass it to `makeRequest` along with the `query` object itself. If all is well, we make a new `json.Decoder` from the body of the request and keep reading inside an infinite `for` loop by calling the `Decode` method. If there is an error (probably due to the connection being closed), we simply break the loop and exit the function. If there is a tweet to read, it will be decoded into the `t` variable, which will give us access to the `Text` property (the 140 characters of the tweet itself). We then iterate over all the possible options, and if the tweet has mentioned it, we send it on the `votes` channel. This technique also allows a tweet to contain many votes at the same time, something you may or may not decide to change based on the rules of the election.



The `votes` channel is send-only (which means we cannot receive on it), since it is of the `chan<- string` type. Think of the little arrow that tells us which way messages will flow: either into the channel (`chan<-`) or out of it (`<-chan`). This is a great way to express intent to other programmers or our future selves—it's clear that we never intend to read `votes` using our `readFromTwitter` function; rather, we will only send them on that channel.

Terminating the program whenever `Decode` returns an error doesn't provide a very robust solution. This is because the Twitter API documentation states that the connection will drop from time to time, and clients should consider this when consuming the services. And remember, we are going to terminate the connection periodically too, so we need to think about a way to reconnect once the connection is dropped.

Signal channels

A great use of channels in Go is to signal events between code running in different goroutines. We are going to see a real-world example of this when we write our next function.

The purpose of the function is to start a goroutine that continually calls the `readFromTwitter` function (with the specified `votes` channel to receive the votes on) until we signal that we want it to stop. And once it has stopped, we want to be notified through another signal channel. The return of the function will be a channel of `struct{}`: a signal channel.

Signal channels have some interesting properties that are worth taking a closer look at. Firstly, the type sent down the channels is an empty `struct{}`, instances of which actually take up zero bytes, since it has no fields. So, `struct{}{}` is a great memory-efficient option for signaling events. Some people use `bool` types, which are also fine, although `true` and `false` both take up a byte of memory.



Head over to <http://play.golang.org> and try this out for yourself. The size of `bool` is one: `fmt.Println(reflect.TypeOf(true).Size()) = 1` On the other hand, the size of `struct{}{}` is zero: `fmt.Println(reflect.TypeOf(struct{}{}).Size()) = 0`

The signal channels also have a buffer size of 1, which means that execution will not get blocked until something reads the signal from the channel.

We are going to employ two signal channels in our code: one that we pass into our function that tells our goroutine that it should stop and another (provided by the function) that signals once the stopping is complete.

In `twitter.go`, add the following function:

```
func startTwitterStream(stopchan <-chan struct{}, votes chan<- string) <-chan struct{} {
    stoppedchan := make(chan struct{}, 1)
    go func() {
        defer func() {
            stoppedchan <- struct{}{}
        }()
        for {
            select {
            case <-stopchan:
                log.Println("stopping Twitter...")
                return
            default:
                log.Println("Querying Twitter...")
                readFromTwitter(votes)
                log.Println("  (waiting)")
                time.Sleep(10 * time.Second) // wait before reconnecting
            }
        }
    }()
    return stoppedchan
}
```

In the preceding code, the first argument, `stopchan`, is a channel of type `<-chan struct{}`, a receive-only signal channel. It is this channel that, outside the code, will signal on, which will tell our goroutine to stop. Remember that it's receive-only inside this function; the actual channel itself will be capable of sending. The second argument is the `votes` channel on which votes will be sent. The return type of our function is also a signal channel of type `<-chan struct{}`: a receive-only channel that we will use to indicate that we have stopped.

These channels are necessary because our function triggers its own goroutine and immediately returns; so without this, calling code would have no idea whether the spawned code was still running or not.

The first thing we do in the `startTwitterStream` function is make our `stoppedchan` argument, and defer the sending of `struct {}{}` to indicate that we have finished when our function exits. Note that `stoppedchan` is a normal channel, so even though it is returned as receive-only, we will be able to send it from within this function.

We then start an infinite `for` loop in which we select from one of two channels. The first is `stopchan` (the first argument), which would indicate that it was time to stop and return (thus triggering the deferred signaling on `stoppedchan`). If that hasn't happened, we will call `readFromTwitter` (passing in the `votes` channel), which will go and load the options from the database and open the connection to Twitter.

When the Twitter connection dies, our code will return, where we sleep for 10 seconds using the `time.Sleep` function. This is to give the Twitter API rest in case it closed the connection due to overuse. Once we've rested, we re-enter the loop and check on `stopchan` again to see whether calling code wants us to stop or not.

To make this flow clear, we are logging out key statements that will not only help us debug our code, but also let us peek into the inner workings of this somewhat complicated mechanism.



Signal channels are a great solution for simple cases where all code lives inside a single package. If you need to cross API boundaries, the `context` package is the recommended way to deal with deadlines, cancelation and, stopping since it was promoted to the standard library in Go 1.7.

Publishing to NSQ

Once our code successfully notices votes on Twitter and sends them down the `votes` channel, we need a way to publish them into an NSQ topic; after all, this is the point of the `twittervotes` program.

We will write a function called `publishVotes`, which will take the `votes` channel, this time of type `<-chan string` (a receive only channel), and publish each string that is received from it.



In our previous functions, the `votes` channel was of type `chan<-string`, but this time, it's of the type `<-chan string`. You might think this is a mistake or even that it means that we cannot use the same channel for both, but you would be wrong. The channel we create later will be made with `make(chan string)`, neither receive nor only send, and can act in both cases. The reason for using the `<-` operator on a channel in



arguments is to make the intent of what the channel will be used for clear, or in the case where it is the return type, to prevent users from accidentally sending on channels intended for receiving or vice versa. The compiler will actually produce an error if they use such a channel incorrectly.

Once the `votes` channel is closed (this is how the external code will tell our function to stop working), we will stop publishing and send a signal down the returned `stop` signal channel.

Add the `publishVotes` function to `main.go`:

```
func publishVotes(votes <-chan string) <-chan struct{} {  
    stopchan := make(chan struct{}, 1)  
    pub, _ := nsq.NewProducer("localhost:4150",  
        nsq.NewConfig())  
    go func() {  
        for vote := range votes {  
            pub.Publish("votes", []byte(vote)) // publish vote  
        }  
        log.Println("Publisher: Stopping")  
        pub.Stop()  
        log.Println("Publisher: Stopped")  
        stopchan <- struct{}{}  
    }()  
    return stopchan  
}
```

Again, the first thing we do is create `stopchan`, which we later return, this time not deferring the signaling but doing it inline by sending `struct{}{}` down `stopchan`.



The difference in how we handle `stopchan` is to show alternative options. Within one code base, you should pick a style you like and stick with it until a standard emerges within the community; in which case, we should all go with that. It is also possible to close `stopchan` rather than send anything down it, which will also unblock the code waiting on that channel. But once a channel is closed, it cannot be reopened.

We then create an NSQ producer by calling `NewProducer` and connecting to the default NSQ port on `localhost` using a default configuration. We start a goroutine, which uses another great built-in feature of the Go language that lets us continually pull values from a channel (in our case, the `votes` channel) just by doing a normal `for...range` operation on it. Whenever the channel has no values, execution will be blocked until one comes down the line. If the `votes` channel is closed, the `for` loop will exit.



To learn more about the power of channels in Go, it is highly recommended that you seek out blog posts and videos by John Graham-Cumming, in particular, one entitled *A Channel Compendium* that he presented at Gophercon 2014 and which contains a brief history of channels, including their origin (interestingly, John was also the guy who successfully petitioned the British government to officially apologize for its treatment of the late, great Alan Turing).

When the loop exits (after the `votes` channel is closed), the publisher is stopped, following which the `stopchan` signal is sent. Did anything stand-out as unusual in the `publishVotes` function? We are breaking a cardinal rule of Go by ignoring an error (assigning it to an underscore variable; therefore dismissing it). As an additional exercise, catch the error and deal with it in a way that seems suitable.

Gracefully starting and stopping programs

When our program is terminated, we want to do a few things before actually exiting, namely closing our connection to Twitter and stopping the NSQ publisher (which actually deregisters its interest in the queue). To achieve this, we have to override the default `Ctrl + C` behavior.



The upcoming code blocks all go inside the `main` function; they are broken up so that we can discuss each section before continuing.

Add the following code inside the `main` function:

```
var stoplock sync.Mutex // protects stop
stop := false
stopChan := make(chan struct{}, 1)
signalChan := make(chan os.Signal, 1)
go func() {
    <-signalChan
    stoplock.Lock()
    stop = true
    stoplock.Unlock()
    log.Println("Stopping...")
    stopChan <- struct{}{}
    closeConn()
}()
signal.Notify(signalChan, syscall.SIGINT, syscall.SIGTERM)
```

Here, we create a stop bool with an associated sync.Mutex function so that we can access it from many goroutines at the same time. We then create two more signal channels, stopChan and signalChan, and use signal.Notify to ask Go to send the signal down signalChan when someone tries to halt the program (either with the SIGINT interrupt or the SIGTERM termination POSIX signals). The stopChan function is how we indicate that we want our processes to terminate, and we pass it as an argument to startTwitterStream later.

We then run a goroutine that blocks waiting for the signal by trying to read from signalChan; this is what the <- operator does in this case (it's trying to read from the channel). Since we don't care about the type of signal, we don't bother capturing the object returned on the channel. Once a signal is received, we set stop to true and close the connection. Only when one of the specified signals is sent will the rest of the goroutine code run, which is how we are able to perform teardown code before exiting the program.

Add the following piece of code (inside the main function) to open and defer the closing of the database connection:

```
if err := dialdb(); err != nil {  
    log.Fatalln("failed to dial MongoDB:", err)  
}  
defer closedb()
```

Since the readFromTwitter method reloads the options from the database each time and because we want to keep our program updated without having to restart it, we are going to introduce one final goroutine. This goroutine will simply call closeConn every minute, causing the connection to die and cause readFromTwitter to be called all over again. Insert the following code at the bottom of the main function to start all of these processes and then wait for them to gracefully stop:

```
// start things  
votes := make(chan string) // chan for votes  
publisherStoppedChan := publishVotes(votes)  
twitterStoppedChan := startTwitterStream(stopChan, votes)  
go func() {  
    for {  
        time.Sleep(1 * time.Minute)  
        closeConn()  
        stoplock.Lock()  
        if stop {  
            stoplock.Unlock()  
            return  
        }  
        stoplock.Unlock()  
    }  
}
```

```
}()  
<-twitterStoppedChan  
close(votes)  
<-publisherStoppedChan
```

First, we make the `votes` channel that we have been talking about throughout this section, which is a simple channel of strings. Note that it is neither a send (`chan<-`) nor a receive (`<-chan`) channel; in fact, making such channels makes little sense. We then call `publishVotes`, passing in the `votes` channel for it to receive from and capturing the returned stop signal channel as `publisherStoppedChan`. Similarly, we call `startTwitterStream`, passing in our `stopChan` function from the beginning of the `main` function and the `votes` channel for it to send to while capturing the resulting stop signal channel as `twitterStoppedChan`.

We then start our refresher goroutine, which immediately enters an infinite `for` loop before sleeping for a minute and closing the connection via the call to `closeConn`. If the `stop bool` has been set to true (in that previous goroutine), we will break the loop and exit; otherwise, we will loop around and wait another minute before closing the connection again. The use of `stoplock` is important because we have two goroutines that might try to access the `stop` variable at the same time, but we want to avoid collisions.

Once the goroutine has started, we block `twitterStoppedChan` by attempting to read from it. When successful (which means the signal was sent on `stopChan`), we close the `votes` channel, which will cause the publisher's `for...range` loop to exit and the publisher itself to stop, after which the signal will be sent on `publisherStoppedChan`, which we wait for before exiting.

Testing

To make sure our program works, we need to do two things: first, we need to create a poll in the database, and second, we need to peer inside the messaging queue to see whether the messages are indeed being generated by `twittervotes`.

In a terminal, run the `mongo` command to open a database shell that allows us to interact with MongoDB. Then, enter the following commands to add a test poll:

```
> use ballots  
switched to db ballots  
> db.polls.insert({ "title": "Test poll", "options":  
  ["happy", "sad", "fail", "win"] })
```

The preceding commands add a new item to the `polls` collection in the `ballots` database. We are using some common words for options that are likely to be mentioned by people on Twitter so that we can observe real tweets being translated into messages. You might notice that our poll object is missing the `results` field; this is fine since we are dealing with unstructured data where documents do not have to adhere to a strict schema. The `counter` program we are going to write in the next section will add and maintain the `results` data for us later.

Press `Ctrl + C` to exit the MongoDB shell and type the following command:

```
nsq_tail --topic="votes" --lookupd-http-
address=localhost:4161
```

The `nsq_tail` tool connects to the specified messaging queue topic and outputs any messages that it notices. This is where we will validate that our `twittervotes` program is sending messages.

In a separate terminal window, let's build and run the `twittervotes` program:

```
go build -o twittervotes
./twittervotes
```

Now switch back to the window running `nsq_tail` and note that messages are indeed being generated in response to live Twitter activity.



If you don't see much activity, try to look up trending hash tags on Twitter and add another poll containing these options.

Counting votes

The second program we are going to implement is the `counter` tool, which will be responsible for watching out for votes in NSQ, counting them, and keeping MongoDB up to date with the latest numbers.

Create a new folder called `counter` alongside `twittervotes`, and add the following code to a new `main.go` file:

```
package main
import (
    "flag"
    "fmt"
```

```
"os"
)
var fatalErr error
func fatal(e error) {
    fmt.Println(e)
    flag.PrintDefaults()
    fatalErr = e
}
func main() {
    defer func() {
        if fatalErr != nil {
            os.Exit(1)
        }
    }()
}
```

Normally when we encounter an error in our code, we use a call such as `log.Fatal` or `os.Exit`, which immediately terminates the program. Exiting the program with a nonzero exit code is important because it is our way of telling the operating system that something went wrong, and we didn't complete our task successfully. The problem with the normal approach is that any deferred functions we have scheduled (and therefore any teardown code we need to run) won't get a chance to execute.

The pattern employed in the preceding code snippet lets us call the `fatal` function to record that an error has occurred. Note that only when our main function exits will the deferred function run, which in turn calls `os.Exit(1)` to exit the program with an exit code of 1. Because the deferred statements are run in LIFO (last in, first out) order, the first function we defer will be the last function to be executed, which is why the first thing we do in the `main` function is defer the exiting code. This allows us to be sure that other functions we defer will be called *before* the program exits. We'll use this feature to ensure that our database connection gets closed regardless of any errors.

Connecting to the database

The best time to think about cleaning up resources, such as database connections, is immediately after you have successfully obtained the resource; Go's `defer` keyword makes this easy. At the bottom of the main function, add the following code:

```
log.Println("Connecting to database...")
db, err := mgo.Dial("localhost")
if err != nil {
    fatal(err)
    return
}
```

```
defer func() {
    log.Println("Closing database connection...")
    db.Close()
}()
pollData := db.DB("ballots").C("polls")
```

This code uses the familiar `mgo.Dial` method to open a session to the locally running MongoDB instance and immediately defers a function that closes the session. We can be sure that this code will run before our previously deferred statement containing the exit code (because deferred functions are run in the reverse order in which they were called). Therefore, whatever happens in our program, we know that the database session will definitely and properly close.



The log statements are optional, but they will help us see what's going on when we run and exit our program.

At the end of the snippet, we use the `mgo` fluent API to keep a reference of the `ballots.polls` data collection in the `pollData` variable, which we will use later to make queries.

Consuming messages in NSQ

In order to count the votes, we need to consume the messages in the `votes` topic in NSQ, and we'll need a place to store them. Add the following variables to the `main` function:

```
var counts map[string]int
var countsLock sync.Mutex
```

A map and a lock (`sync.Mutex`) is a common combination in Go because we will have multiple goroutines trying to access the same map, and we need to avoid corrupting it by trying to modify or read it at the same time.

Add the following code to the `main` function:

```
log.Println("Connecting to nsq...")
q, err := nsq.NewConsumer("votes", "counter", nsq.NewConfig())
if err != nil {
    fatal(err)
    return
}
```

The `NewConsumer` function allows us to set up an object that will listen on the `votes` NSQ topic, so when `twittervotes` publishes a vote on that topic, we can handle it in this program. If `NewConsumer` returns an error, we'll use our `fatal` function to record it and return.

Next, we are going to add the code that handles messages (`votes`) from NSQ:

```
q.AddHandler(nsq.HandlerFunc(func(m *nsq.Message) error {
    countsLock.Lock()
    defer countsLock.Unlock()
    if counts == nil {
        counts = make(map[string]int)
    }
    vote := string(m.Body)
    counts[vote]++
    return nil
}))
```

We call the `AddHandler` method on `nsq.Consumer` and pass it a function that will be called for every message received on the `votes` topic.

When a vote comes in, the first thing we do is lock the `countsLock` mutex. Next, we defer the unlocking of the mutex for when the function exits. This allows us to be sure that while `NewConsumer` is running, we are the only ones allowed to modify the map; others will have to wait until our function exits before they can use it. Calls to the `Lock` method block execution while the lock is in place, and it only continues when the lock is released by a call to `Unlock`. This is why it's vital that every `Lock` call has an `Unlock` counterpart; otherwise, we will deadlock our program.

Every time we receive a vote, we check whether `counts` is `nil` and make a new map if it is because once the database has been updated with the latest results, we want to reset everything and start at zero. Finally, we increase the `int` value by one for the given key and return `nil`, indicating no errors.

Although we have created our NSQ consumer and added our handler function, we still need to connect to the NSQ service, which we will do by adding the following code:

```
if err := q.ConnectToNSQLookupd("localhost:4161");
err != nil {
    fatal(err)
    return
}
```



It is important to note that we are actually connecting to the HTTP port of the nsqlookupd instance rather than NSQ instances; this abstraction means that our program doesn't need to know *where* the messages are coming from in order to consume them. If we fail to connect to the server (for instance, if we forget to start it), we'll get an error, which we report to our fatal function before immediately returning.

Keeping the database updated

Our code will listen out for votes and keep a map of the results in the memory, but that information is trapped inside our program so far. Next, we need to add the code that will periodically push the results to the database. Add the following `doCount` function:

```
func doCount(countsLock *sync.Mutex, counts *map[string]int, pollData
*mgo.Collection) {
    countsLock.Lock()
    defer countsLock.Unlock()
    if len(*counts) == 0 {
        log.Println("No new votes, skipping database update")
        return
    }
    log.Println("Updating database...")
    log.Println(*counts)
    ok := true
    for option, count := range *counts {
        sel := bson.M{"options": bson.M{"$in":
            []string{option}}}
        up := bson.M{"$inc": bson.M{"results." +
            option:count}}
        if _, err := pollData.UpdateAll(sel, up); err != nil {
            log.Println("failed to update:", err)
            ok = false
        }
    }
    if ok {
        log.Println("Finished updating database...")
        *counts = nil // reset counts
    }
}
```

When our `doCount` function runs, the first thing we do is lock `countsLock` and defer its unlocking. We then check to see whether there are any values in the `counts` map. If there aren't, we just log that we're skipping the update and wait for next time.

We are taking all arguments in as pointers (note the `*` character before the type name) because we want to be sure that we are interacting with the underlying data itself and not a copy of it. For example, the `*counts = nil` line will actually reset the underlying map to `nil` rather than just invalidate our local copy of it. If there are some votes, we iterate over the `counts` map, pulling out the option and the number of votes (since the last update), and use some MongoDB magic to update the results.



MongoDB stores **BSON** (short for **Binary JSON**) documents internally, which are easier to traverse than normal JSON documents, and that is why the `mgo` package comes with the `mgo/bson` encoding package. When using `mgo`, we will often use `bson` types, such as the `bson.M` map, to describe concepts for MongoDB.

We first create the selector for our update operation using the `bson.M` shortcut type, which is similar to creating `map[string]interface{}` types. The selector we create here will look something like this:

```
{  
    "options": {  
        "$in": ["happy"]  
    }  
}
```

In MongoDB, the preceding BSON specifies that we want to select polls where "happy" is one of the items in the `options` array.

Next, we use the same technique to generate the update operation, which looks something like this:

```
{  
    "$inc": {  
        "results.happy": 3  
    }  
}
```

In MongoDB, the preceding BSON specifies that we want to increase the `results.happy` field by three. If there is no `results` map in the poll, one will be created, and if there is no `happy` key inside `results`, zero will be assumed.

We then call the `UpdateAll` method in our `pollsData` query to issue the command to the database, which will in turn update every poll that matches the selector (contrast this to the `Update` method, which will update only one). If something goes wrong, we report it and set the `ok` Boolean to `false`. If all goes well, we set the `counts` map to `nil`, since we want to reset the counter.

We are going to specify `updateDuration` as a constant at the top of the file, which will make it easy for us to change when we are testing our program. Add the following code above the `main` function:

```
const updateDuration = 1 * time.Second
```

Next, we will add `time.Ticker` and make sure our `doCount` function gets called in the same `select` block that we use when responding to *Ctrl + C*.

Responding to Ctrl + C

The last thing to do before our program is ready is set up a `select` block that periodically calls `doCount` and be sure that our `main` function waits for operations to complete before exiting, like we did in our `twittervotes` program. Add the following code at the end of the `main` function:

```
ticker := time.NewTicker(updateDuration)
termChan := make(chan os.Signal, 1)
signal.Notify(termChan, syscall.SIGINT, syscall.SIGTERM, syscall.SIGHUP)
for {
    select {
    case <-ticker.C:
        doCount(&countsLock, &counts, pollData)    case <- termChan:ticker.Stop()
        q.Stop()
    case <-q.StopChan:
        // finished
        return
    }
}
```

The `time.Ticker` function is a type that gives us a channel (via the `c` field) on which the current time is sent at the specified interval (in our case, `updateDuration`). We use this in a `select` block to call our `doCount` function while `termChan` and `q.StopChan` are quiet.

To handle termination, we have employed a slightly different tactic than before. We trap the termination event, which will cause a signal to go down `termChan` when we hit *Ctrl + C*. Next, we start an infinite loop, inside which we use the `select` structure to allow us to run the code if we receive something on either `termChan` or `StopChan` of the consumer.

In fact, we will only ever get a `termChan` signal first in response to a `Ctrl + C` press, at which point we stop `time.Ticker` and ask the consumer to stop listening for votes. Execution then re-enters the loop and blocks until the consumer reports that it has indeed stopped by signaling on its `StopChan` function. When that happens, we're done and we exit, at which point our deferred statement runs, which, if you remember, tidies up the database session.

Running our solution

It's time to see our code in action. Ensure that you have `nsqlookupd`, `nsqd`, and `mongod` running in separate terminal windows with the following:

```
nsqlookupd
nsqd --lookupd-tcp-address=127.0.0.1:4160
mongod --dbpath ./db
```

If you haven't already done so, make sure the `twittervotes` program is running too. Then, in the `counter` folder, build and run our counting program:

```
go build -o counter
./counter
```

You should see a periodic output describing what work `counter` is doing, such as the following:

```
No new votes, skipping database update
Updating database...
map[win:2 happy:2 fail:1]
Finished updating database...
No new votes, skipping database update
Updating database...
map[win:3]
Finished updating database...
```

The output will, of course, vary since we are actually responding to real, live activity on Twitter.



We can see that our program is receiving vote data from NSQ and reports to update the database with the results. We can confirm this by opening the MongoDB shell and querying the poll data to see whether the `results` map is being updated. In another terminal window, open the MongoDB shell:

```
mongo
```

Ask it to use the ballots database:

```
> use ballots
switched to db ballots
```

Use the `find` method with no arguments to get all polls (add the `pretty` method to the end to get nicely formatted JSON):

```
> db.polls.find().pretty()
{
  "_id" : ObjectId("53e2a3afffbff195c2e09a02"),
  "options" : [
    "happy", "sad", "fail", "win"
  ],
  "results" : {
    "fail" : 159, "win" : 711,
    "happy" : 233, "sad" : 166,
  },
  "title" : "Test poll"
}
```

The `results` map is indeed updated, and at any point in time, it contains the total number of votes for each option.

Summary

In this chapter, we covered a lot of ground. We learned different techniques to gracefully shut down programs using signaling channels, which is especially important when our code has some work to do before it can exit. We saw that deferring the reporting of fatal errors at the start of our program can give our other deferred functions a chance to execute before the process ends.

We also discovered how easy it is to interact with MongoDB using the `mgo` package and how to use BSON types when describing concepts for the database. The `bson.M` alternative to `map[string]interface{}` helps us keep our code more concise while still providing all the flexibility we need when working with unstructured or schemaless data.

We learned about message queues and how they allow us to break apart the components of a system into isolated and specialized micro-services. We started an instance of NSQ by first running the `nsqlookupd` lookup daemon before running a single `nsqd` instance and connecting them via a TCP interface. We were then able to publish votes to the queue in `twittervotes` and connect to the lookup daemon to run a handler function for every vote sent in our `counter` program.

While our solution is actually performing a pretty simple task, the architecture we have put together in this chapter is capable of doing some pretty great things.

We eliminated the need for our `twittervotes` and `counter` programs to run on the same machine-as long as they can both connect to the appropriate NSQ, they will function as expected regardless of where they are running.

We can distribute our MongoDB and NSQ nodes across many physical machines, which would mean our system is capable of gigantic scale-whenver resources start running low, we can add new boxes to cope with the demand.

When we add other applications that need to query and read the results from polls, we can be sure that our database services are highly available and capable of delivering.

We can spread our database across geographical expanses, replicating data for backup so we don't lose anything when disaster strikes.

We can build a multinode, fault-tolerant NSQ environment, which means that when our `twittervotes` program learns of interesting tweets, there will always be some place to send the data.

We can write many more programs that generate votes from different sources; the only requirement is that they know how to put messages into NSQ.

In the next chapter, we will build a RESTful data service of our own, through which we will expose the functionality of our social polling application. We will also build a web interface that lets users create their own polls and have the results visualized.

6

Exposing Data and Functionality through a RESTful Data Web Service API

In the previous chapter, we built a service that reads tweets from Twitter, counts the hash tag votes, and stores the results in a MongoDB database. We also used the MongoDB shell to add polls and see the poll results. This approach is fine if we are the only ones using our solution, but it would be madness if we released our project and expected users to connect directly to our MongoDB instance in order to use the service we built.

Therefore, in this chapter, we are going to build a RESTful data service through which the data and functionality will be exposed. We will also put together a simple website that consumes the new API. Users may then either use our website to create and monitor polls or build their own application on top of the web services we release.



Code in this chapter depends on the code in [Chapter 5, Building Distributed Systems and Working with Flexible Data](#), so it is recommended that you complete that chapter first, especially since it covers setting up the environment that the code in this chapter runs on.

Specifically, you will learn:

- How wrapping `http.HandlerFunc` types can give us a simple but powerful pipeline of execution for our HTTP requests
- How to safely share data between HTTP handlers using the `context` package
- Best practices for the writing of handlers responsible for exposing data

- Where small abstractions can allow us to write the simplest possible implementations now but leave room to improve them later without changing the interface
- How adding simple helper functions and types to our project will prevent us from (or at least defer) adding dependencies on external packages

RESTful API design

For an API to be considered RESTful, it must adhere to a few principles that stay true to the original concepts behind the Web and are already known to most developers. Such an approach allows us to make sure we aren't building anything strange or unusual into our API while also giving our users a head start toward consuming it, since they are already familiar with its concepts.

Some of the important RESTful design concepts are:

- HTTP methods describe the kind of action to take; for example, GET methods will only ever read data, while POST requests will create something
- Data is expressed as a collection of resources
- Actions are expressed as changes to data
- URLs are used to refer to specific data
- HTTP headers are used to describe the kind of representation coming into and going out of the server

The following table shows the HTTP methods and URLs that represent the actions that we will support in our API, along with a brief description and an example use case of how we intend the call to be used.

Request	Description	Use case
GET /polls	Read all polls	Show a list of polls to the users
GET /polls/{id}	Read the poll	Show details or results of a specific poll
POST /polls	Create a poll	Create a new poll
DELETE /polls/{id}	Delete a poll	Delete a specific poll

The {id} placeholder represents where in the path the unique ID for a poll will go.

Sharing data between handlers

Occasionally, we need to share a state between our middleware and handlers. Go 1.7 brought the `context` package into the standard library, which gives us, among other things, a way to share basic request-scoped data.

Every `http.Request` method comes with a `context.Context` object accessible via the `request.Context()` method, from which we can create new context objects. We can then call `request.WithContext()` to get a (cheap) shallow copied `http.Request` method that uses our new `Context` object.

To add a value, we can create a new context (based on the existing one from the request) via the `contextWithValue` method:

```
ctx := contextWithValue(r.Context(), "key", "value")
```



While you can technically store any type of data using this approach, it is only recommended that you store simple primitive types such as Strings and Integers and do not use it to inject dependencies or pointers to other objects that your handlers might need. Later in this chapter, we will explore patterns to access dependencies, such as a database connection.

In middleware code, we can then use our new `ctx` object when we pass execution to the wrapped handler:

```
Handler.ServeHTTP(w, r.WithContext(ctx))
```

It is worth exploring the documentation for the `context` package at <https://golang.org/pkg/context/> in order to find out what other features it provides.

We are going to use this technique to allow our handlers to have access to an API key that is extracted and validated elsewhere.

Context keys

Setting a value in a context object requires us to use a key, and while it might seem obvious that the `value` argument is of type `interface{}`, which means we can (but not necessarily should) store anything we like, it might surprise you to learn the type of the key:

```
func WithValue(parent Context, key, val interface{}) Context
```

The key is also an `interface{}`. This means we are not restricted to using only strings as the key, which is good news when you consider how disparate code might well attempt to set values with the same name in the same context, which would create problems.

Instead, a pattern of a more stable way of keying values is emerging from the Go community (and is already used in some places inside the standard library). We are going to create a simple (private) struct for our keys and a helper method in order to get the value out of the context.

Add the essential minimal `main.go` file inside a new `api` folder:

```
package main
func main() {}
```

Add a new type called `contextKey`:

```
type contextKey struct {
    name string
}
```

This structure contains only the name of the key, but pointers to it will remain unique even if the name field is the same in two keys. Next, we are going to add a key to store our API key value in:

```
var contextKeyAPIKey = &contextKey{"api-key"}
```

It is good practice to group related variables together with a common prefix; in our case, we can start the name all of our context key types with the `contextKey` prefix. Here, we have created a key called `contextKeyAPIKey`, which is a pointer to a `contextKey` type, setting the name as `api-key`.

Next, we are going to write a helper that will, given a context, extract the key:

```
func APIKey(ctx context.Context) (string, bool) {
    key, ok := ctx.Value(contextKeyAPIKey).(string)
    return key, ok
}
```

The function takes `context.Context` and returns the API key string along with an `ok` bool indicating whether the key was successfully obtained and cast to a string or not. If the key is missing, or if it's the wrong type, the second return argument will be false, but our code will not panic.

Note that `contextKey` and `contextKeyAPIKey` are internal (they start with a lowercase letter) but `APIKey` will be exported. In `main` packages, this doesn't really matter, but if you were writing a package, it's nice to know that the complexity of how you are storing and extracting data from a context is hidden from users.

Wrapping handler functions

We are going to utilize one of the most valuable patterns to learn when building services and websites in Go, something we already explored a little in [Chapter 2, Adding User Accounts](#): wrapping handlers. We have seen how we can wrap `http.Handler` types to run code before and after our main handlers execute, and we are going to apply the same technique to `http.HandlerFunc` function alternatives.

API keys

Most web APIs require clients to register an API key for their application, which they are asked to send along with every request. Such keys have many purposes, ranging from simply identifying which app the requests are coming from to addressing authorization concerns in situations where some apps are only able to do limited things based on what a user has allowed. While we don't actually need to implement API keys for our application, we are going to ask clients to provide one, which will allow us to add an implementation later, while keeping the interface constant.

We are going to add our first `HandlerFunc` wrapper function called `withAPIKey` to the bottom of `main.go`:

```
func withAPIKey(fn http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        key := r.URL.Query().Get("key")
        if !isValidAPIKey(key) {
            respondErr(w, r, http.StatusUnauthorized, "invalid
                API key")
            return
        }
        ctx := context.WithValue(r.Context(),
            contextKeyAPIKey, key)
        fn(w, r.WithContext(ctx))
    }
}
```

As you can see, our `withAPIKey` function both takes an `http.HandlerFunc` type as an argument and returns one; this is what we mean by wrapping in this context. The `withAPIKey` function relies on a number of other functions that we are yet to write, but you can clearly see what's going on. Our function immediately returns a new `http.HandlerFunc` type that performs a check for the `key` query parameter by calling `isValidAPIKey`. If the key is deemed invalid (by the return of `false`), we respond with an invalid API key error; otherwise, we put the key into the context and call the next handler. To use this wrapper, we simply pass an `http.HandlerFunc` type into this function in order to enable the `key` parameter check. Since it returns an `http.HandlerFunc` type too, the result can then be passed on to other wrappers or given directly to the `http.HandleFunc` function to actually register it as the handler for a particular path pattern.

Let's add our `isValidAPIKey` function next:

```
func isValidAPIKey(key string) bool {  
    return key == "abc123"  
}
```

For now, we are simply going to hardcode the API key as `abc123`; anything else will return `false` and therefore be considered invalid. Later, we can modify this function to consult a configuration file or database to check the authenticity of a key without affecting how we use the `isValidAPIKey` method or the `withAPIKey` wrapper.

Cross-origin resource sharing

The same-origin security policy mandates that AJAX requests in web browsers be allowed only for services hosted on the same domain, which would make our API fairly limited since we won't necessarily be hosting all of the websites that use our web service. The **CORS (Cross-origin resource sharing)** technique circumnavigates the same-origin policy, allowing us to build a service capable of serving websites hosted on other domains. To do this, we simply have to set the `Access-Control-Allow-Origin` header in response to `*`. While we're at it, since we're going to use the `Location` header in our create poll call – we'll allow this header to be accessible by the client too, which can be done by listing it in the `Access-Control-Expose-Headers` header. Add the following code to `main.go`:

```
func withCORS(fn http.HandlerFunc) http.HandlerFunc {  
    return func(w http.ResponseWriter, r *http.Request) {  
        w.Header().Set("Access-Control-Allow-Origin", "*")  
        w.Header().Set("Access-Control-Expose-Headers",  
            "Location")  
        fn(w, r)  
    }  
}
```

```
}
```

This is the simplest wrapper function yet; it just sets the appropriate header on the `ResponseWriter` type and calls the specified `http.HandlerFunc` type.



In this chapter, we are handling CORS explicitly so we can understand exactly what is going on; for real production code, you should consider employing an open source solution, such as <https://github.com/fasterness/cors>.

Injecting dependencies

Now that we can be sure that a request has a valid API key and is CORS-compliant, we must consider how handlers will connect to the database. One option is to have each handler dial its own connection, but this isn't very **DRY (Don't Repeat Yourself)** and leaves room for potentially erroneous code, such as code that forgets to close a database session once it is finished with it. It also means that if we wanted to change how we connected to the database (perhaps we want to use a domain name instead of a hardcoded IP address), we might have to modify our code in many places, rather than one.

Instead, we will create a new type that encapsulates all the dependencies for our handlers and construct it with a database connection in `main.go`.

Create a new type called `Server`:

```
// Server is the API server.
type Server struct {
    db *mgo.Session
}
```

Our handler functions will be methods of this server, which is how they will be able to access the database session.

Responding

A big part of any API is responding to requests with a combination of status codes, data, errors, and sometimes headers – the `net/http` package makes all of this very easy to do. One option we have, which remains the best option for tiny projects or even the early stages of big projects, is to just build the response code directly inside the handler.

As the number of handlers grows, however, we will end up duplicating a lot of code and sprinkling representation decisions all over our project. A more scalable approach is to abstract the response code into helper functions.

For the first version of our API, we are going to speak only JSON, but we want the flexibility to add other representations later if we need to.

Create a new file called `respond.go` and add the following code:

```
func decodeBody(r *http.Request, v interface{}) error {
    defer r.Body.Close()
    return json.NewDecoder(r.Body).Decode(v)
}
func encodeBody(w http.ResponseWriter, r *http.Request, v interface{}) error {
    return json.NewEncoder(w).Encode(v)
}
```

These two functions abstract the decoding and encoding of data from and to the `Request` and `ResponseWriter` objects, respectively. The decoder also closes the request body, which is recommended. Although we haven't added much functionality here, it means that we do not need to mention JSON anywhere else in our code, and if we decide to add support for other representations or switch to a binary protocol instead, we only need to touch these two functions.

Next, we are going to add a few more helpers that will make responding even easier. In `respond.go`, add the following code:

```
func respond(w http.ResponseWriter, r *http.Request,
    status int, data interface{}) {
    w.WriteHeader(status)
    if data != nil {
        encodeBody(w, r, data)
    }
}
```

This function makes it easy to write the status code and some data to the `ResponseWriter` object using our `encodeBody` helper.

Handling errors is another important aspect that is worth abstracting. Add the following `respondErr` helper:

```
func respondErr(w http.ResponseWriter, r *http.Request,
    status int, args ...interface{}) {
    respond(w, r, status, map[string]interface{}{
        "error": map[string]interface{}{}}
```

```
        "message": fmt.Sprint(args...),  
    },  
}  
}
```

This method gives us an interface similar to the `respond` function, but the data written will be enveloped in an `error` object in order to make it clear that something went wrong. Finally, we can add an HTTP-error-specific helper that will generate the correct message for us using the `http.StatusText` function from the Go standard library:

```
func respondHTTPErr(w http.ResponseWriter, r *http.Request, status int) {  
    respondErr(w, r, status, http.StatusText(status))  
}
```

Note that these functions are all dog food, which means that they use each other (as in, eating your own dog food), which is important since we want actual responding to happen in only one place for if (or more likely, when) we need to make changes.

Understanding the request

The `http.Request` object gives us access to every piece of information we might need about the underlying HTTP request; therefore, it is worth glancing through the `net/http` documentation to really get a feel for its power. Examples include, but are not limited to, the following:

- The URL, path, and query string
- The HTTP method
- Cookies
- Files
- Form values
- The referrer and user agent of requester
- Basic authentication details
- The request body
- The header information

There are a few things it doesn't address, which we need to either solve ourselves or look to an external package to help us with. URL path parsing is one such example – while we can access a path (such as `/people/1/books/2`) as a string via the `http.Request` type's `URL.Path` field, there is no easy way to pull out the data encoded in the path, such as the people ID of `1` or the book ID of `2`.



A few projects do a good job of addressing this problem, such as Goweb or Gorillz's `mux` package. They let you map path patterns that contain placeholders for values that they then pull out of the original string and make available to your code. For example, you can map a pattern of `/users/{userID}/comments/{commentID}`, which will map paths such as `/users/1/comments/2`. In your handler code, you can then get the values by the names placed inside the curly braces rather than having to parse the path yourself.

Since our needs are simple, we are going to knock together a simple path-parsing utility; we can always use a different package later if we have to, but that would mean adding a dependency to our project.

Create a new file called `path.go` and insert the following code:

```
package main
import (
    "strings"
)
const PathSeparator = "/"
type Path struct {
    Path string
    ID   string
}
func NewPath(p string) *Path {
    var id string
    p = strings.Trim(p, PathSeparator)
    s := strings.Split(p, PathSeparator)
    if len(s) > 1 {
        id = s[len(s)-1]
        p = strings.Join(s[:len(s)-1], PathSeparator)
    }
    return &Path{Path: p, ID: id}
}
func (p *Path) HasID() bool {
    return len(p.ID) > 0
}
```

This simple parser provides a `NewPath` function that parses the specified path string and returns a new instance of the `Path` type. Leading and trailing slashes are trimmed (using `strings.Trim`) and the remaining path is split (using `strings.Split`) by the `PathSeparator` constant, which is just a forward slash. If there is more than one segment (`len(s) > 1`), the last one is considered to be the ID. We re-slice the slice of strings to select the last item for the ID using `s[len(s)-1]` and the rest of the items for the remainder of the path using `s[:len(s)-1]`. On the same lines, we also rejoin the path segments with the `PathSeparator` constant to form a single string containing the path without the ID.

This supports any `collection/id` pair, which is all we need for our API. The following table shows the state of the `Path` type for the given original path string:

Original path string	Path	ID	HasID
/	/	nil	false
/people/	people	nil	false
/people/1/	people	1	true

Serving our API with one function

A web service is nothing more than a simple Go program that binds to a specific HTTP address and port and serves requests, so we get to use all our command-line tool writing knowledge and techniques.



We also want to ensure that our `main` function is as simple and modest as possible, which is always a goal of coding, especially in Go.

Before writing our `main` function, let's look at a few design goals of our API program:

- We should be able to specify the HTTP address and port to which our API listens and the address of the MongoDB instances without having to recompile the program (through command-line flags)

We want the program to gracefully shut down when we terminate it, allowing the in-flight requests (requests that are still being processed when the termination signal is sent to our program) to complete

- We want the program to log out status updates and report errors properly

Atop the `main.go` file, replace the `main` function placeholder with the following code:

```
func main() {  
    var (  
        addr = flag.String("addr", ":8080", "endpoint  
        address")  
        mongo = flag.String("mongo", "localhost", "mongodb  
        address")  
    )  
    log.Println("Dialing mongo", *mongo)  
    db, err := mgo.Dial(*mongo)  
    if err != nil {  
        log.Fatalln("failed to connect to mongo:", err)  
    }  
    defer db.Close()  
    s := &Server{  
        db: db,  
    }  
    mux := http.NewServeMux()  
    mux.HandleFunc("/polls/",  
        withCORS(withAPIKey(s.handlePolls)))  
    log.Println("Starting web server on", *addr)  
    http.ListenAndServe(":8080", mux)  
    log.Println("Stopping...")  
}
```

This function is the entirety of our API `main` function. The first thing we do is specify two command-line flags, `addr` and `mongo`, with some sensible defaults and ask the `flag` package to parse them. We then attempt to dial the MongoDB database at the specified address. If we are unsuccessful, we abort with a call to `log.Fatalln`. Assuming the database is running and we are able to connect, we store the reference in the `db` variable before deferring the closing of the connection. This ensures that our program properly disconnects and tidies up after itself when it ends.



We create our server and specify the database dependency. We are calling our server `s`, which some people think is a bad practice because it's difficult to read code referring to a single letter variable and know what it is. However, since the scope of this variable is so small, we can be sure that its use will be very near to its definition, removing the potential for confusion.

We then create a new `http.ServeMux` object, which is a request multiplexer provided by the Go standard library, and register a single handler for all requests that begin with the `/polls/` path. Note that the `handlePolls` handler is a method on our server, and this is how it will be able to access the database.

Using handler function wrappers

It is when we call `HandleFunc` on the `ServeMux` handler that we are making use of our handler function wrappers with this line:

```
withCORS(withAPIKey(handlePolls))
```

Since each function takes an `http.HandlerFunc` type as an argument and also returns one, we are able to chain the execution just by nesting the function calls, as we have done previously. So when a request comes in with a path prefix of `/polls/`, the program will take the following execution path:

1. The `withCORS` function is called, which sets the appropriate header.
2. The `withAPIKey` function is called next, which checks the request for an API key and aborts if it's invalid or else calls the next handler function.
3. The `handlePolls` function is then called, which may use the helper functions in `respond.go` to write a response to the client.
4. Execution goes back to `withAPIKey`, which exits.
5. Execution finally goes back to `withCORS`, which exits.

Handling endpoints

The final piece of the puzzle is the `handlePolls` function, which will use the helpers to understand the incoming request and access the database and generate a meaningful response that will be sent back to the client. We also need to model the poll data that we were working with in the previous chapter.

Create a new file called `polls.go` and add the following code:

```
package main
import "gopkg.in/mgo.v2/bson"
type poll struct {
    ID      bson.ObjectId `bson:"_id" json:"id"`
    Title   string        `json:"title"`
    Options []string      `json:"options"`
```

```
Results map[string]int `json:"results,omitempty"`
APIKey string `json:"apikey"`
}
```

Here, we define a structure called `poll`, which has five fields that in turn describe the polls being created and maintained by the code we wrote in the previous chapter. We have also added the `APIKey` field, which you probably wouldn't do in the real world but which will allow us to demonstrate how we extract the API key from the context. Each field also has a tag (two in the `ID` case), which allows us to provide some extra metadata.

Using tags to add metadata to structs

Tags are just a string that follows a field definition within a `struct` type on the same line. We use the black tick character to denote literal strings, which means we are free to use double quotes within the tag string itself. The `reflect` package allows us to pull out the value associated with any key; in our case, both `bson` and `json` are examples of keys, and they are each key/value pair separated by a space character. Both the `encoding/json` and `gopkg.in/mgo.v2/bson` packages allow you to use tags to specify the field name that will be used with encoding and decoding (along with some other properties) rather than having it infer the values from the name of the fields themselves. We are using BSON to talk with the MongoDB database and JSON to talk to the client, so we can actually specify different views of the same `struct` type. For example, consider the `ID` field:

```
ID bson.ObjectId `bson:"_id" json:"id"`


```

The name of the field in Go is `ID`, the JSON field is `id`, and the BSON field is `_id`, which is the special identifier field used in MongoDB.

Many operations with a single handler

Because our simple path-parsing solution cares only about the path, we have to do some extra work when looking at the kind of RESTful operation the client is making. Specifically, we need to consider the HTTP method so that we know how to handle the request. For example, a `GET` call to our `/polls/` path should read `polls`, where a `POST` call would create a new one. Some frameworks solve this problem for you by allowing you to map handlers based on more than the path, such as the HTTP method or the presence of specific headers in the request. Since our case is ultra simple, we are going to use a simple `switch` case.

In `polls.go`, add the `handlePolls` function:

```
func (s *Server) handlePolls(w http.ResponseWriter,
    r *http.Request) {
```

```
switch r.Method {
    case "GET":
        s.handlePollsGet(w, r)
        return
    case "POST":
        s.handlePollsPost(w, r)
        return
    case "DELETE":
        s.handlePollsDelete(w, r)
        return
}
// not found
respondHTTPErr(w, r, http.StatusNotFound)
}
```

We switch on the HTTP method and branch our code depending on whether it is GET, POST, or DELETE. If the HTTP method is something else, we just respond with a 404 `http.StatusNotFound` error. To make this code compile, you can add the following function stubs underneath the `handlePolls` handler:

```
func (s *Server) handlePollsGet(w http.ResponseWriter,
    r *http.Request) {
    respondErr(w, r, http.StatusInternalServerError,
        errors.New("not
            implemented"))
}
func (s *Server) handlePollsPost(w http.ResponseWriter,
    r *http.Request) {
    respondErr(w, r, http.StatusInternalServerError,
        errors.New("not
            implemented"))
}
func (s *Server) handlePollsDelete(w http.ResponseWriter,
    r *http.Request) {
    respondErr(w, r, http.StatusInternalServerError,
        errors.New("not
            implemented"))
}
```



In this section, we learned how to manually parse elements of the requests (the HTTP method) and make decisions in code. This is great for simple cases, but it's worth looking at packages such as Gorilla's `mux` package for some more powerful ways of solving these problems. Nevertheless, keeping external dependencies to a minimum is a core philosophy of writing good and contained Go code.

Reading polls

Now it's time to implement the functionality of our web service. Add the following code:

```
func (s *Server) handlePollsGet(w http.ResponseWriter,
    r *http.Request) {
    session := s.db.Copy()
    defer session.Close()
    c := session.DB("ballots").C("polls")
    var q *mgo.Query
    p := NewPath(r.URL.Path)
    if p.HasID() {
        // get specific poll
        q = c.FindId(bson.ObjectIdHex(p.ID))
    } else {
        // get all polls
        q = c.Find(nil)
    }
    var result []*poll
    if err := q.All(&result); err != nil {
        respondErr(w, r, http.StatusInternalServerError, err)
        return
    }
    respond(w, r, http.StatusOK, &result)
}
```

The very first thing we do in each of our sub handler functions is create a copy of the database session that will allow us to interact with MongoDB. We then use `mgo` to create an object referring to the `polls` collection in the database – if you remember, this is where our `polls` live.

We then build up an `mgo.Query` object by parsing the path. If an ID is present, we use the `FindId` method on the `polls` collection; otherwise, we pass `nil` to the `Find` method, which indicates that we want to select all the `polls`. We are converting the ID from a string to a `bson.ObjectId` type with the `ObjectIdHex` method so that we can refer to the `polls` with their numerical (hex) identifiers.

Since the `All` method expects to generate a collection of `poll` objects, we define the `result` as `[] *poll` or a slice of pointers to `poll` types. Calling the `All` method on the query will cause `mgo` to use its connection to MongoDB to read all the `polls` and populate the `result` object.



For small scale, such as a small number of polls, this approach is fine, but as the polls grow, we will need to consider a more sophisticated approach. We can page the results by iterating over them using the `Iter` method on the query and using the `Limit` and `Skip` methods, so we do not try to load too much data into the memory or present too much information to users in one go.

Now that we have added some functionality, let's try out our API for the first time. If you are using the same MongoDB instance that we set up in the previous chapter, you should already have some data in the `polls` collection; to see our API working properly, you should ensure there are at least two polls in the database.

If you need to add other polls to the database, in a terminal, run the `mongo` command to open a database shell that will allow you to interact with MongoDB. Then, enter the following commands to add some test polls:

```
> use ballots
switched to db ballots
> db.polls.insert({ "title": "Test poll", "options": ["one", "two", "three"] })
> db.polls.insert({ "title": "Test poll two", "options": ["four", "five", "six"] })
```

In a terminal, navigate to your `api` folder and build and run the project:

```
go build -o api
./api
```

Now make a GET request to the `/polls/` endpoint by navigating to `http://localhost:8080/polls/?key=abc123` in your browser; remember to include the trailing slash. The result will be an array of polls in the JSON format.

Copy and paste one of the IDs from the polls list and insert it before the `?` character in the browser to access the data for a specific poll, for example, `http://localhost:8080/polls/5415b060a02cd4adb487c3ae?key=abc123`. Note that instead of returning all the polls, it only returns one.



Test the API key functionality by removing or changing the key parameter to see what the error looks like.

You might have also noticed that although we are only returning a single poll, this poll value is still nested inside an array. This is a deliberate design decision made for two reasons: the first and most important reason is that nesting makes it easier for users of the API to write code to consume the data. If users are always expecting a JSON array, they can write strong types that describe that expectation rather than having one type for single polls and another for collections of polls. As an API designer, this is your decision to make. The second reason we left the object nested in an array is that it makes the API code simpler, allowing us to just change the `mgo.Query` object and leave the rest of the code the same.

Creating a poll

Clients should be able to make a POST request to `/polls/` in order to create a poll. Let's add the following code inside the POST case:

```
func (s *Server) handlePollsPost(w http.ResponseWriter,
    r *http.Request) {
    session := s.db.Copy()
    defer session.Close()
    c := session.DB("ballots").C("polls")
    var p poll
    if err := decodeBody(r, &p); err != nil {
        respondErr(w, r, http.StatusBadRequest, "failed to
            read poll from request", err)
        return
    }
    apikey, ok := APIKey(r.Context())
    if ok {
        p.APIKey = apikey
    }
    p.ID = bson.NewObjectId()
    if err := c.Insert(p); err != nil {
        respondErr(w, r, http.StatusInternalServerError,
            "failed to insert
            poll", err)
        return
    }
    w.Header().Set("Location", "polls/" + p.ID.Hex())
    respond(w, r, http.StatusCreated, nil)
}
```

After we get a copy of the database session like earlier, we attempt to decode the body of the request that, according to RESTful principles, should contain a representation of the poll object the client wants to create. If an error occurs, we use the `respondErr` helper to write the error to the user and immediately exit from the function. We then generate a new unique ID for the poll and use the `mgo` package's `Insert` method to send it into the database. We then set the `Location` header of the response and respond with a `201 http.StatusCreated` message, pointing to the URL from which the newly created poll may be accessed. Some APIs return the object instead of providing a link to it; there is no concrete standard so it's up to you as the designer.

Deleting a poll

The final piece of functionality we are going to include in our API is the ability to delete polls. By making a request with the `DELETE` HTTP method to the URL of a poll (such as `/polls/5415b060a02cd4adb487c3ae`), we want to be able to remove the poll from the database and return a `200 Success` response:

```
func (s *Server) handlePollsDelete(w http.ResponseWriter,
    r *http.Request) {
    session := s.db.Copy()
    defer session.Close()
    c := session.DB("ballots").C("polls")
    p := NewPath(r.URL.Path)
    if !p.HasID() {
        respondErr(w, r, http.StatusMethodNotAllowed,
            "Cannot delete all polls.")
        return
    }
    if err := c.RemoveId(bson.ObjectIdHex(p.ID)); err != nil {
        respondErr(w, r, http.StatusInternalServerError,
            "failed to delete poll", err)
        return
    }
    respond(w, r, http.StatusOK, nil) // ok
}
```

Similar to the GET case, we parse the path, but this time, we respond with an error if the path does not contain an ID. For now, we don't want people to be able to delete all polls with one request, and so we use the suitable `StatusMethodNotAllowed` code. Then, using the same collection we used in the previous cases, we call `RemoveId`, passing the ID in the path after converting it into a `bson.ObjectId` type. Assuming things go well, we respond with an `http.StatusOK` message with no body.

CORS support

In order for our `DELETE` capability to work over CORS, we must do a little extra work to support the way CORS browsers handle some HTTP methods such as `DELETE`. A CORS browser will actually send a preflight request (with an HTTP method of `OPTIONS`), asking for permission to make a `DELETE` request (listed in the `Access-Control-Request-Method` request header), and the API must respond appropriately in order for the request to work. Add another case in the switch statement for `OPTIONS`:

```
case "OPTIONS":  
    w.Header().Add("Access-Control-Allow-Methods", "DELETE")  
    respond(w, r, http.StatusOK, nil)  
    return
```

If the browser asks for permission to send a `DELETE` request, the API will respond by setting the `Access-Control-Allow-Methods` header to `DELETE`, thus overriding the default `*` value that we set in our `withCORS` wrapper handler. In the real world, the value for the `Access-Control-Allow-Methods` header will change in response to the request made, but since `DELETE` is the only case we are supporting, we can hardcode it for now.



The details of CORS are out of the scope of this book, but it is recommended that you research the particulars online if you intend to build truly accessible web services and APIs. Head over to <http://enable-cors.org/> to get started.

Testing our API using curl

Curl is a command-line tool that allows us to make HTTP requests to our service so that we can access it as though we were a real app or client consuming the service.



Windows users do not have access to curl by default and will need to seek an alternative. Check out <http://curl.haxx.se/dlwiz/?type=bin> or search the Web for Windows curl alternative.

In a terminal, let's read all the polls in the database through our API. Navigate to your api folder and build and run the project and also ensure MongoDB is running:

```
go build -o api  
./api
```

We then perform the following steps:

1. Enter the following curl command that uses the -X flag to denote we want to make a GET request to the specified URL:

```
curl -X GET http://localhost:8080/polls/?  
key=abc123
```

2. The output is printed after you hit *Enter*:

```
[{"id": "541727b08ea48e5e5d5bb189", "title": "Best  
Beatle?",  
"options": ["john", "paul", "george", "ringo"]},  
{"id": "541728728ea48e5e5d5bb18a", "title": "Favorite  
language?",  
"options": ["go", "java", "javascript", "ruby"]}]
```

3. While it isn't pretty, you can see that the API returns the polls from your database. Issue the following command to create a new poll:

```
curl --data '{"title": "test", "options":  
["one", "two", "three"]}'  
-X POST http://localhost:8080/polls/?key=abc123
```

4. Get the list again to see the new poll included:

```
curl -X GET http://localhost:8080/polls/?  
key=abc123
```

5. Copy and paste one of the IDs and adjust the URL to refer specifically to that poll:

```
curl -X GET  
http://localhost:8080/polls/541727b08ea48e5e5d5bb189?  
key=abc123  
[{"id": "541727b08ea48e5e5d5bb189", "title": "Best Beatle?",  
"options": ["john", "paul", "george", "ringo"]}]
```

6. Now we see only the selected poll. Let's make a `DELETE` request to remove the poll:

```
curl -X DELETE  
http://localhost:8080/polls/541727b08ea48e5e5d5bb189?  
key=abc123
```

7. Now when we get all the polls again, we'll see that the Beatles poll has gone:

```
curl -X GET http://localhost:8080/polls/?key=abc123  
[{"id": "541728728ea48e5e5d5bb18a", "title": "Favorite  
language?", "options": ["go", "java", "javascript", "ruby"]}]
```

So now that we know that our API is working as expected, it's time to build something that consumes the API properly.

A web client that consumes the API

We are going to put together an ultra simple web client that consumes the capabilities and data exposed through our API, allowing users to interact with the polling system we built in the previous chapter and earlier in this chapter. Our client will be made up of three web pages:

- An `index.html` page that shows all the polls
- A `view.html` page that shows the results of a specific poll
- A `new.html` page that allows users to create new polls

Create a new folder called `web` alongside the `api` folder and add the following content to the `main.go` file:

```
package main  
import (  
    "flag"  
    "log"  
    "net/http"  
)  
func main() {  
    var addr = flag.String("addr", ":8081", "website address")  
    flag.Parse()  
    mux := http.NewServeMux()  
    mux.Handle("/", http.StripPrefix("/",  
        http.FileServer(http.Dir("public"))))  
    log.Println("Serving website at:", *addr)
```

```
    http.ListenAndServe(*addr, mux)
}
```

These few lines of Go code really highlight the beauty of the language and the Go standard library. They represent a complete, highly scalable, static website hosting program. The program takes an `addr` flag and uses the familiar `http.ServeMux` type to serve static files from a folder called `public`.



Building the next few pages –while we're building the UI –consists of writing a lot of HTML and JavaScript code. Since this is not Go code, if you'd rather not type it all out, feel free to head over to the GitHub repository for this book and copy and paste it from <https://github.com/matryer/goblueprints>. You are also free to include the latest versions of the Bootstrap and jQuery libraries as you see fit, but there may be implementation differences with subsequent versions.

Index page showing a list of polls

Create the `public` folder inside `web` and add the `index.html` file after writing the following HTML code in it:

```
<!DOCTYPE html>
<html>
<head>
  <title>Polls</title>
  <link rel="stylesheet"
    href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/
    bootstrap.min.css">
</head>
<body>
</body>
</html>
```

We will use Bootstrap again to make our simple UI look nice, but we need to add two additional sections to the `body` tag of the HTML page. First, add the DOM elements that will display the list of polls:

```
<div class="container">
  <div class="col-md-4"></div>
  <div class="col-md-4">
    <h1>Polls</h1>
    <ul id="polls"></ul>
    <a href="new.html" class="btn btn-primary">Create new poll</a>
  </div>
```

```
<div class="col-md-4"></div>
</div>
```

Here, we are using Bootstrap's grid system to center-align our content that is made up of a list of polls and a link to new.html, where users can create new polls.

Next, add the following script tags and JavaScript underneath that:

```
<script
src="//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js"></script>
<script>
$(function() {
    var update = function() {
        $.get("http://localhost:8080/polls/?key=abc123", null, null, "json")
            .done(function(polls) {
                $("#polls").empty();
                for (var p in polls) {
                    var poll = polls[p];
                    $("#polls").append(
                        $(<li>)
                            .append($(<a>)
                                .attr("href", "view.html?poll=polls/" + poll.id)
                                .text(poll.title)
                            )
                    );
                }
            });
        window.setTimeout(update, 10000);
    }
    update();
});
</script>
```

We are using jQuery's `$.get` function to make an AJAX request to our web service. We are hardcoding the API URL—which, in practice, you might decide against—or at least use a domain name to abstract it. Once the polls have loaded, we use jQuery to build up a list containing hyperlinks to the `view.html` page, passing the ID of the poll as a query parameter.

Creating a new poll

To allow users to create a new poll, create a file called `new.html` inside the `public` folder, and add the following HTML code to the file:

```
<!DOCTYPE html>
<html>
<head>
    <title>Create Poll</title>
    <link rel="stylesheet"
        href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/
        bootstrap.min.css">
</head>
<body>
    <script src="//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js">
</script>
</body>
</html>
```

We are going to add the elements for an HTML form that will capture the information we need when creating a new poll, namely the title of the poll and the options. Add the following code inside the `body` tags:

```
<div class="container">
    <div class="col-md-4"></div>
    <form id="poll" role="form" class="col-md-4">
        <h2>Create Poll</h2>
        <div class="form-group">
            <label for="title">Title</label>
            <input type="text" class="form-control" id="title"
                placeholder="Title">
        </div>
        <div class="form-group">
            <label for="options">Options</label>
            <input type="text" class="form-control" id="options"
                placeholder="Options">
            <p class="help-block">Comma separated</p>
        </div>
        <button type="submit" class="btn btn-primary">
            Create Poll</button> or <a href="/">cancel</a>
    </form>
    <div class="col-md-4"></div>
</div>
```

Since our API speaks JSON, we need to do a bit of work to turn the HTML form into a JSON-encoded string and also break the comma-separated options string into an array of options. Add the following script tag:

```
<script>
$(function(){
    var form = $("form#poll");
    form.submit(function(e){
        e.preventDefault();
        var title = form.find("input[id='title']").val();
        var options = form.find("input[id='options']").val();
        options = options.split(",");
        for (var opt in options) {
            options[opt] = options[opt].trim();
        }
        $.post("http://localhost:8080/polls/?key=abc123",
            JSON.stringify({
                title: title, options: options
            })
        ).done(function(d, s, r){
            location.href = "view.html?poll=" +
            r.getResponseHeader("Location");
        });
    });
});
</script>
```

Here, we add a listener to the `submit` event of our form and use jQuery's `val` method to collect the input values. We split the options with a comma and trim the spaces away before using the `$.post` method to make the `POST` request to the appropriate API endpoint. `JSON.stringify` allows us to turn the data object into a JSON string, and we use that string as the body of the request, as expected by the API. On success, we pull out the `Location` header and redirect the user to the `view.html` page, passing a reference to the newly created poll as the parameter.

Showing the details of a poll

The final page of our app we need to complete is the `view.html` page, where users can see the details and live results of the poll. Create a new file called `view.html` inside the `public` folder and add the following HTML code to it:

```
<!DOCTYPE html>
<html>
<head>
```

```
<title>View Poll</title>
<link rel="stylesheet"
      href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
</head>
<body>
<div class="container">
  <div class="col-md-4"></div>
  <div class="col-md-4">
    <h1 data-field="title">...</h1>
    <ul id="options"></ul>
    <div id="chart"></div>
    <div>
      <button class="btn btn-sm" id="delete">Delete this poll</button>
    </div>
  </div>
  <div class="col-md-4"></div>
</div>
</body>
</html>
```

This page is mostly similar to the other pages; it contains elements to present the title of the poll, the options, and a pie chart. We will be mashing up Google's Visualization API with our API to present the results. Underneath the final `div` tag in `view.html` (and above the closing `body` tag), add the following `script` tags:

```
<script src="//www.google.com/jsapi"></script>
<script src="//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js">
</script>
<script>
google.load('visualization', '1.0', {'packages':['corechart']});
google.setOnLoadCallback(function() {
  $(function(){
    var chart;
    var poll = location.href.split("poll=")[1];
    var update = function(){
      $.get("http://localhost:8080/" + poll + "?key=abc123", null, null,
            "json")
        .done(function(polls){
          var poll = polls[0];
          $('[data-field="title"]').text(poll.title);
          $('#options').empty();
          for (var o in poll.results) {
            $('#options').append(
              $('- ').append(
                $('').addClass("label label-default").text(poll.results[o]),
                " ", o
              )
            )
          }
        })
    }
  });
});

```

```
        )
    }
    if (poll.results) {
        var data = new google.visualization.DataTable();
        data.addColumn("string", "Option");
        data.addColumn("number", "Votes");
        for (var o in poll.results) {
            data.addRow([o, poll.results[o]])
        }
        if (!chart) {
            chart = new google.visualization.PieChart
                (document.getElementById('chart'));
        }
        chart.draw(data, {is3D: true});
    }
}
);
window.setTimeout(update, 1000);
};

update();
$("#delete").click(function(){
    if (confirm("Sure?")) {
        $.ajax({
            url:"http://localhost:8080/"+poll+"?key=abc123",
            type:"DELETE"
        })
        .done(function(){
            location.href = "/";
        })
    }
});
});
});
});
</script>
```

We include the dependencies we will need in order to power our page, jQuery and Bootstrap, and also the Google JavaScript API. The code loads the appropriate visualization libraries from Google and waits for the DOM elements to load before extracting the poll ID from the URL by splitting it on `poll=`. We then create a variable called `update` that represents a function responsible for generating the view of the page. This approach is taken to make it easy for us to use `window.setTimeout` in order to issue regular calls to update the view. Inside the `update` function, we use `$.get` to make a GET request to our `/polls/{id}` endpoint, replacing `{id}` with the actual ID we extracted from the URL earlier. Once the poll has loaded, we update the title on the page and iterate over the options to add them to the list. If there are results (remember, in the previous chapter, the `results` map was only added to the data as votes started being counted), we create a new

`google.visualization.PieChart` object and build a `google.visualization.DataTable` object containing the results. Calling `draw` on the chart causes it to render the data and thus update the chart with the latest numbers. We then use `setTimeout` to tell our code to call `update` again in another second.

Finally, we bind to the `click` event of the delete button we added to our page, and after asking the user whether they are sure, make a `DELETE` request to the polls URL and then redirect them back to the home page. It is this request that will actually cause the `OPTIONS` request to be made first, asking for permission, which is why we added explicit support for it in our `handlePolls` function earlier.

Running the solution

We built many components over the previous two chapters, and it is now time to see them all working together. This section contains everything you need in order to get all the items running, assuming you have the environment set up properly, as described at the beginning of the previous chapter. This section assumes you have a single folder that contains the four subfolders: `api`, `counter`, `twittervotes`, and `web`.

Assuming nothing is running, take the following steps (each step in its own terminal window):

1. In the top-level folder, start the `nsqlookupd` daemon:

```
nsqlookupd
```

2. In the same directory, start the `nsqd` daemon:

```
nsqd --lookupd-tcp-address=localhost:4160
```

3. Start the MongoDB daemon:

```
mongod
```

4. Navigate to the `counter` folder and build and run it:

```
cd counter
go build -o counter
./counter
```

5. Navigate to the `twittervotes` folder and build and run it. Ensure that you have the appropriate environment variables set; otherwise, you will see errors when you run the program:

```
cd ../twittervotes  
go build -o twittervotes  
./twittervotes
```

6. Navigate to the `api` folder and build and run it:

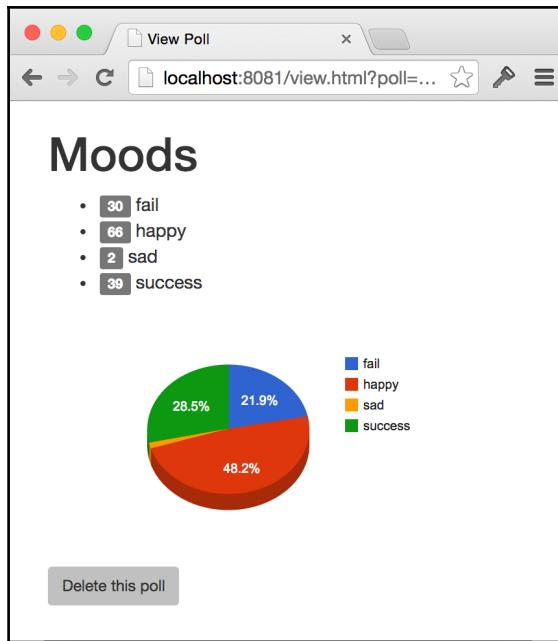
```
cd ../api  
go build -o api  
./api
```

7. Navigate to the `web` folder and build and run it:

```
cd ../web  
go build -o web  
./web
```

Now that everything is running, open a browser and head to `http://localhost:8081/`. Using the user interface, create a poll called `Moods` and input the options as `happy, sad, fail, success`. These are common enough words that we are likely to see some relevant activity on Twitter.

Once you have created your poll, you will be taken to the view page where you will start to see the results coming in. Wait for a few seconds and enjoy the fruits of your hard work as the UI updates in real time, showing live, real-time results:



Summary

In this chapter, we exposed the data for our social polling solution through a highly scalable RESTful API and built a simple website that consumes the API to provide an intuitive way for users to interact with it. The website consists of static content only, with no server-side processing (since the API does the heavy lifting for us). This allows us to host the website very cheaply on static hosting sites, such as bitballoon.com, or distribute the files to content delivery networks.

Within our API service, we learned how to share data between handlers without breaking or obfuscating the handler pattern from the standard library. We also saw how writing wrapped handler functions allows us to build a pipeline of functionality in a very simple and intuitive way.

We wrote some basic encoding and decoding functions that –while only simply wrapping their counterparts from the `encoding/json` package for now –could be improved later to support a range of different data representations without changing the internal interface to our code. We wrote a few simple helper functions that make responding to data requests easy while providing the same kind of abstraction that would allow us to evolve our API later.

We saw how, for simple cases, switching to HTTP methods is an elegant way to support many functions for a single endpoint. We also saw how, with a few extra lines of code, we are able to build support for CORS in order to allow applications running on different domains to interact with our services –without the need for hacks such as JSONP.

In the next chapter, we will evolve our API and web skills to build a brand new startup app called Meander. We'll also explore an interesting way of representing enumerators in a language that doesn't officially support them.

7

Random Recommendations Web Service

The concept behind the project that we will build in this chapter is a simple one: we want users to be able to generate random recommendations for things to do in specific geographical locations based on a predefined set of journey types that we will expose through the API. We will give our project the codename Meander.

Often on projects in the real world, you are not responsible for the full stack; somebody else builds the website, a different person might write the iOS app, and maybe an outsourced company builds the desktop version. On more successful API projects, you might not even know who the consumers of your API are, especially if it's a public API.

In this chapter, we will simulate this reality by designing and agreeing a minimal API design with a fictional partner up front before going on to implement the API. Once we have finished our side of the project, we will download a user interface built by our teammates to see the two work together to produce the final application.

In this chapter, you will:

- Learn to express the general goals of a project using short and simple Agile user stories
- Discover that you can agree on a meeting point in a project by agreeing on the design of an API, which allows many people to work in parallel
- See how early versions can have data fixtures written in code and compiled into the program, allowing us to change the implementation later without touching the interface
- Learn a strategy that allows structs (and other types) to represent a public version of themselves for cases where we want to hide or transform internal representations

- Learn to use embedded structs to represent nested data while keeping the interface of our types simple
- Learn to use `http.Get` to make external API requests, specifically to the Google Places API, with no code bloat
- Learn to effectively implement enumerators in Go even though they aren't really a language feature
- Experience a real-world example of TDD
- Look at how the `math/rand` package makes it easy to select an item from a slice at random
- Learn an easy way to grab data from the URL parameters of the `http.Request` type

The project overview

Following Agile methodologies, let's write two user stories that describe the functionality of our project. User stories shouldn't be comprehensive documents describing the entire set of features of an application; rather, small cards are perfect for not only describing what the user is trying to do, but also why. Also, we should do this without trying to design the whole system up front or delving too deep into implementation details.

First, we need a story about seeing the different journey types from which our users can select:

As a	traveler
I want	to see the different types of journeys I can get recommendations for
So that	I can decide what kind of evening to take my partner on

Secondly, we need a story about providing random recommendations for a selected journey type:

As a	traveler
I want	to see a random recommendation for my selected journey type
So that	I know where to go and what the evening will entail

These two stories represent the two core capabilities that our API needs to provide and actually ends up representing two endpoints.

In order to discover places around specified locations, we are going to make use of the Google Places API, which allows us to search for listings of businesses with given types, such as `bar`, `cafe`, or `movie_theater`. We will then use Go's `math/rand` package to pick from these places at random, building up a complete journey for our users.



The Google Places API supports many business types; refer to https://developers.google.com/places/documentation/supported_types for the complete list.

Project design specifics

In order to turn our stories into an interactive application, we are going to provide two JSON endpoints: one to deliver the kinds of journeys users will be able to select in the application and another to actually generate the random recommendations for the selected journey type.

```
GET /journeys
```

The preceding call should return a list similar to the following:

```
[  
  {  
    name: "Romantic",  
    journey: "park|bar|movie_theater|restaurant|florist"  
  },  
  {  
    name: "Shopping",  
    journey: "department_store|clothing_store|jewelry_store"  
  }  
]
```

The `name` field is a human-readable label for the type of recommendations the app generates, and the `journey` field is a pipe-separated list of the supported journey types. It is the `journey` value that we will pass, as a URL parameter, into our other endpoint, which generates the actual recommendations:

```
GET /recommendations?  
lat=1&lng=2&journey=bar|cafe&radius=10&cost=$....$$$$
```

This endpoint is responsible for querying the Google Places API and generating the recommendations before returning an array of place objects. We will use the parameters in the URL to control the kind of query to make. The `lat` and `lng` parameters representing latitude and longitude, respectively tell our API where in the world we want recommendations from, and the `radius` parameter represents the distance in meters around the point in which we are interested.

The `cost` value is a human-readable way of representing the price range for places that the API returns. It is made up of two values: a lower and upper range separated by three dots. The number of dollar characters represents the price level, with `$` being the most affordable and `$$$$$` being the most expensive. Using this pattern, a value of `$...$` would represent very low-cost recommendations, where `$$$$...$$$$$` would represent a pretty expensive experience.



Some programmers might insist that the cost range is represented by numerical values, but since our API is going to be consumed by people, why not make things a little more interesting? It is up to you as the API designer.

An example payload for this call might look something like this:

```
[  
 {  
   icon: "http://maps.gstatic.com/mapfiles/place_api/icons/cafe-  
    71.png",  
   lat: 51.519583, lng: -0.146251,  
   vicinity: "63 New Cavendish St, London",  
   name: "Asia House",  
   photos: [{  
     url: "https://maps.googleapis.com/maps/api/place/photo?  
      maxwidth=400&photoreference=CnRnAAAAyLRN"  
   }]  
 }, ...  
 ]
```

The array returned contains a place object representing a random recommendation for each segment in the journey in the appropriate order. The preceding example is a cafe in London. The data fields are fairly self-explanatory; the `lat` and `lng` fields represent the location of the place, the `name` and `vicinity` fields tell us what and where the business is, and the `photos` array gives us a list of relevant photographs from Google's servers. The `vicinity` and `icon` fields will help us deliver a richer experience to our users.

Representing data in code

We are first going to expose the journeys that users can select from; so, create a new folder called `meander` in `GOPATH` and add the following `journeys.go` code:

```
package meander
type j struct {
    Name      string
    PlaceTypes []string
}
var Journeys = []interface{}{
    j{Name: "Romantic", PlaceTypes: []string{"park", "bar",
        "movie_theater", "restaurant", "florist", "taxi_stand"}},
    j{Name: "Shopping", PlaceTypes: []string{"department_store", "cafe",
        "clothing_store", "jewelry_store", "shoe_store"}},
    j{Name: "Night Out", PlaceTypes: []string{"bar", "casino", "food",
        "bar", "night_club", "bar", "bar", "hospital"}},
    j{Name: "Culture", PlaceTypes: []string{"museum", "cafe", "cemetery",
        "library", "art_gallery"}},
    j{Name: "Pamper", PlaceTypes: []string{"hair_care", "beauty_salon",
        "cafe", "spa"}},
}
```

Here, we define an internal type called `j` inside the `meander` package, which we then use to describe the journeys by creating instances of them inside the `Journeys` slice. This approach is an ultra-simple way of representing data in the code without building a dependency on an external data store.



As an additional assignment, why not see if you can keep `golint` happy throughout this process? Every time you add some code, run `golint` for the packages and satisfy any suggestions that emerge. It cares a lot about exported items that have no documentation; so adding simple comments in the correct format will keep it happy. To learn more about `golint`, refer to <https://github.com/golang/lint>.

Of course, this is likely to evolve into just that later, maybe even with the ability for users to create and share their own journeys. Since we are exposing our data via an API, we are free to change the internal implementation without affecting the interface, so this approach is great for a version 1.



We are using a slice of type `[]interface{}` because we will later implement a general way of exposing public data regardless of the actual types.

A romantic journey consists of a visit first to a park, then a bar, a movie theater, then a restaurant before a visit to a florist, and finally, a taxi ride home; you get the general idea. Feel free to get creative and add others by consulting the supported types in the Google Places API.

You might have noticed that since we are containing our code inside a package called `meander` (rather than `main`), our code can never be run as a tool like the other APIs we have written so far. Create two new folders inside `meander` so that you have a path that looks like `meander/cmd/meander`; this will house the actual command-line tool that exposes the `meander` package's capabilities via an HTTP endpoint.

Since we are primarily building a package for our meandering project (something that other tools can import and make use of), the code in the root folder is the `meander` package, and we nest our command (the `main` package) inside the `cmd` folder. We include the additional final `meander` folder to follow good practices where the command name is the same as the folder if we omitted it, our command would be called `cmd` instead of `meander`, which would get confusing.

Inside the `cmd/meander` folder, add the following code to the `main.go` file:

```
package main
func main() {
    //meander.APIKey = "TODO"
    http.HandleFunc("/journeys", func(w http.ResponseWriter,
        r *http.Request) {
        respond(w, r, meander.Journeys)
    })
    http.ListenAndServe(":8080", http.DefaultServeMux)
}
func respond(w http.ResponseWriter, r *http.Request, data []interface{}) error {
    return json.NewEncoder(w).Encode(data)
}
```

You will recognize this as a simple API endpoint program, mapping to the `/journeys` endpoint.



You'll have to import the `encoding/json`, `net/http`, and `runtime` packages, along with your own `meander` package you created earlier.

We set the value of `APIKey` in the `meander` package (which is commented out for now, since we are yet to implement it) before calling the familiar `HandleFunc` function on the `net/http` package to bind our endpoint, which then just responds with the `meander.Journeys` variable. We borrow the abstract responding concept from the previous chapter by providing a `respond` function that encodes the specified data to the `http.ResponseWriter` type.

Let's run our API program by navigating to the `cmd/meander` folder in a terminal and using `go run`. We don't need to build this into an executable file at this stage since it's just a single file:

```
go run main.go
```

Hit the `http://localhost:8080/journeys` endpoint, and note that our `Journeys` data payload is served, which looks like this:

```
[{  
    Name: "Romantic",  
    PlaceTypes: [  
        "park",  
        "bar",  
        "movie_theater",  
        "restaurant",  
        "florist",  
        "taxi_stand"  
    ]  
, ...]
```

This is perfectly acceptable, but there is one major flaw: it exposes internals about our implementation. If we changed the `PlaceTypes` field name to `Types`, promises made in our API would break, and it's important that we avoid this.

Projects evolve and change over time, especially successful ones, and as developers, we should do what we can to protect our customers from the impact of the evolution. Abstracting interfaces is a great way to do this, as is taking ownership of the public-facing view of our data objects.

Public views of Go structs

In order to control the public view of structs in Go, we need to invent a way to allow individual journey types to tell us how they want to be exposed. In the root `meander` folder, create a new file called `public.go` and add the following code:

```
package meander
type Facade interface {
    Public() interface{}
}
func Public(o interface{}) interface{} {
    if p, ok := o.(Facade); ok {
        return p.Public()
    }
    return o
}
```

The `Facade` interface exposes a single `Public` method, which will return the public view of a struct. The exported `Public` function takes any object and checks whether it implements the `Facade` interface (does it have a `Public() interface{}` method?); if it is implemented, it calls the method and returns the result otherwise, it just returns the original object untouched. This allows us to pass anything through the `Public` function before writing the result to the `ResponseWriter` object, allowing individual structs to control their public appearance.



Normally, single method interfaces such as our `Facade` are named after the method they describe, such as `Reader` and `Writer`. However, `Publicer` is just confusing, so I deliberately broke the rule.

Let's implement a `Public` method for our `j` type by adding the following code to `journeys.go`:

```
func (j j) Public() interface{} {
    return map[string]interface{}{
        "name": j.Name,
        "journey": strings.Join(j.PlaceTypes, "|"),
    }
}
```

The public view of our `j` type joins the `PlaceTypes` field into a single string separated by the pipe character as per our API design.

Head back to `cmd/meander/main.go` and replace the `respond` method with one that makes use of our new `Public` function:

```
func respond(w http.ResponseWriter, r *http.Request, data []interface{}) error {
    publicData := make([]interface{}, len(data))
    for i, d := range data {
        publicData[i] = meander.Public(d)
    }
    return json.NewEncoder(w).Encode(publicData)
}
```

Here, we iterate over the data slice calling the `meander.Public` function for each item, building the results into a new slice of the same size. In the case of our `j` type, its `Public` method will be called to serve the public view of the data rather than the default view. In a terminal, navigate to the `cmd/meander` folder again and run `go run main.go` before hitting `http://localhost:8080/journeys`. Note that the same data has now changed to a new structure:

```
[{
    journey: "park|bar|movie_theater|restaurant|florist|taxi_stand",
    name: "Romantic"
}, ...]
```



An alternative way of achieving the same result would be to use tags to control the field names, as we have done in previous chapters, and implement your own `[]string` type that provides a `MarshalJSON` method which tells the encoder how to marshal your type. Both are perfectly acceptable, but the Facade interface and `Public` method are probably more expressive (if someone reads the code, isn't it obvious what's going on?) and give us more control.

Generating random recommendations

In order to obtain the places from which our code will randomly build up recommendations, we need to query the Google Places API. In the root `meander` folder, add the following `query.go` file:

```
package meander
type Place struct {
    *googleGeometry `json:"geometry"`
    Name            string      `json:"name"`
    Icon            string      `json:"icon"`

    // Additional fields for Place
    Address        string      `json:"address"`
    Geometry       googleGeometry `json:"geometry"`
    Id             string      `json:"id"`
    Latitude       float64     `json:"latitude"`
    Longitude      float64     `json:"longitude"`
    OpenHours      string      `json:"open_hours"`
    Rating         float64     `json:"rating"`
    Reference     string      `json:"reference"`
    Types          []string    `json:"types"`
}
```

```
Photos          [] *googlePhoto `json:"photos"`
Vicinity       string      `json:"vicinity"`
}

type googleResponse struct {
    Results [] *Place `json:"results"`
}

type googleGeometry struct {
    *googleLocation `json:"location"`
}

type googleLocation struct {
    Lat float64 `json:"lat"`
    Lng float64 `json:"lng"`
}

type googlePhoto struct {
    PhotoRef string `json:"photo_reference"`
    URL      string `json:"url"`
}
```

This code defines the structures we will need in order to parse the JSON response from the Google Places API into usable objects.



Head over to the Google Places API documentation for an example of the response we are expecting. Refer to <http://developers.google.com/places/documentation/search>.

Most of the preceding code will be obvious, but it's worth noting that the `Place` type embeds the `googleGeometry` type, which allows us to represent the nested data as per the API while essentially flattening it in our code. We do this with `googleLocation` inside `googleGeometry`, which means that we will be able to access the `Lat` and `Lng` values directly on a `Place` object even though they're technically nested in other structures.

Because we want to control how a `Place` object appears publically, let's give this type the following `Public` method:

```
func (p *Place) Public() interface{} {
    return map[string]interface{}{
        "name":      p.Name,
        "icon":      p.Icon,
        "photos":    p.Photos,
        "vicinity":  p.Vicinity,
        "lat":       p.Lat,
        "lng":       p.Lng,
    }
}
```



Remember to run `golint` on this code to see which comments need to be added to the exported items.

The Google Places API key

Like with most APIs, we will need an API key in order to access the remote services. Head over to the Google APIs Console, sign in with a Google account, and create a key for the Google Places API. For more detailed instructions, refer to the documentation on the Google's developer website.

Once you have your key, let's create a variable inside the `meander` package that can hold it. At the top of `query.go`, add the following definition:

```
var APIKey string
```

Now nip back into `main.go`, remove the double slash `//` from the `APIKey` line, and replace the `TODO` value with the actual key provided by the Google APIs Console. Remember that it is bad practice to hardcode keys like this directly in your code; instead, it's worth breaking them out into environment variables, which keeps them out of your source code repository.

Enumerators in Go

To handle the various cost ranges for our API, it makes sense to use an enumerator (or `enum`) to denote the various values and handle conversions to and from string representations. Go doesn't explicitly provide enumerators as a language feature, but there is a neat way of implementing them, which we will explore in this section.

A simple flexible checklist to write enumerators in Go is as follows:

- Define a new type based on a primitive integer type
- Use that type whenever you need users to specify one of the appropriate values
- Use the `iota` keyword to set the values in a `const` block, disregarding the first zero value
- Implement a map of sensible string representations to the values of your enumerator

- Implement a `String` method on the type that returns the appropriate string representation from the map
- Implement a `ParseType` function that converts from a string to your type using the map

Now, we will write an enumerator to represent the cost levels in our API. Create a new file called `cost_level.go` inside the root `meander` folder and add the following code:

```
package meander
type Cost int8
const (
    _Cost = iota
    Cost1
    Cost2
    Cost3
    Cost4
    Cost5
)
```

Here, we define the type of our enumerator, which we have called `Cost`, and since we need to represent a only few values, we have based it on an `int8` range. For enumerators where we need larger values, you are free to use any of the integer types that work with `iota`. The `Cost` type is now a real type in its own right, and we can use it wherever we need to represent one of the supported values for example, we can specify a `Cost` type as an argument in functions, or we can use it as the type for a field in a struct.

We then define a list of constants of that type and use the `iota` keyword to indicate that we want incrementing values for the constants. By disregarding the first `iota` value (which is always zero), we indicate that one of the specified constants must be explicitly used rather than the zero value.

To provide a string representation of our enumerator, we only need to add a `String` method to the `Cost` type. This is a useful exercise even if you don't need to use the strings in your code, because whenever you use the print calls from the Go standard library (such as `fmt.Println`), the numerical values will be used by default. Often, these values are meaningless and will require you to look them up and even count the lines to determine the numerical value for each item.



For more information on the `String()` method in Go, refer to the `Stringer` and `GoStringer` interfaces in the `fmt` package at <http://golang.org/pkg/fmt/#Stringer>.

Test-driven enumerator

To ensure that our enumerator code is working correctly, we are going to write unit tests that make some assertions about expected behavior.

Alongside `cost_level.go`, add a new file called `cost_level_test.go` and add the following unit test:

```
package meander_test
import (
    "testing"
    "github.com/cheekybits/is"
    "path/to/meander"
)
func TestCostValues(t *testing.T) {
    is := is.New(t)
    is.Equal(int(meander.Cost1), 1)
    is.Equal(int(meander.Cost2), 2)
    is.Equal(int(meander.Cost3), 3)
    is.Equal(int(meander.Cost4), 4)
    is.Equal(int(meander.Cost5), 5)
}
```

You will need to run `go get` in order to get the CheekyBits `is` package (from <https://github.com/cheekybits/is>).



The `is` package is an alternative testing helper package, but this one is ultra-simple and deliberately bare-bones. You get to pick your favorite when you write your own projects or use none at all.

Normally, we wouldn't worry about the actual integer value of constants in our enumerator, but since the Google Places API uses numerical values to represent the same thing, we need to care about the values.



You might have noticed something strange about this test file that breaks from convention. Although it is inside the root `meander` folder, it is not a part of the `meander` package; rather, it's in `meander_test`. In Go, this is an error in every case except for tests. Because we are putting our test code into its own package, it means that we no longer have access to the internals of the `meander` package. Note how we have to use the package prefix. This may seem like a disadvantage, but in fact, it allows us to be sure that we are testing the package as though we were a real user of it. We may only call exported methods and only have visibility into exported types; just like our users. And we cannot mess around with internals to do



things that our users cannot; it's a true user test. In testing, sometimes you do need to fiddle with an internal state, in which case your tests would need to be in the same package as the code instead.

Run the tests by running `go test` in a terminal and note that it passes.

Let's add another test to make assertions about the string representations for each `Cost` constant. In `cost_level_test.go`, add the following unit test:

```
func TestCostString(t *testing.T) {
    is := is.New(t)
    is.Equal(meander.Cost1.String(), "$")
    is.Equal(meander.Cost2.String(), "$$")
    is.Equal(meander.Cost3.String(), "$$$")
    is.Equal(meander.Cost4.String(), "$$$$")
    is.Equal(meander.Cost5.String(), "$$$$$")
}
```

This test asserts that calling the `String` method for each constant yields the expected value. Running these tests will, of course, fail because we haven't implemented the `String` method yet.

Underneath the `Cost` constants, add the following map and the `String` method:

```
var costStrings = map[string]Cost{
    "$":     Cost1,
    "$$":    Cost2,
    "$$$":   Cost3,
    "$$$$":  Cost4,
    "$$$$":  Cost5,
}
func (l Cost) String() string {
    for s, v := range costStrings {
        if l == v {
            return s
        }
    }
    return "invalid"
}
```

The `map[string]Cost` variable maps the cost values to the string representation, and the `String` method iterates over the map to return the appropriate value.



In our case, a simple `strings.Repeat("$", int(l))` return would work just as well (and wins because it's simpler code); but it often won't; therefore, this section explores the general approach.

Now if we were to print out the `Cost3` value, we would actually see `$$$`, which is much more useful than numerical values. As we want to use these strings in our API, we are also going to add a `ParseCost` method.

In `cost_value_test.go`, add the following unit test:

```
func TestParseCost(t *testing.T) {
    is := is.New(t)
    is.Equal(meander.Cost1, meander.ParseCost(""))
    is.Equal(meander.Cost2, meander.ParseCost("$$"))
    is.Equal(meander.Cost3, meander.ParseCost("$$$"))
    is.Equal(meander.Cost4, meander.ParseCost("$$$$"))
    is.Equal(meander.Cost5, meander.ParseCost("$$$$$"))
}
```

Here, we assert that calling `ParseCost` will, in fact, yield the appropriate value depending on the input string.

In `cost_value.go`, add the following implementation code:

```
func ParseCost(s string) Cost {
    return costStrings[s]
}
```

Parsing a `Cost` string is very simple since this is how our map is laid out.

As we need to represent a range of cost values, let's imagine a `CostRange` type and write the tests out for how we intend to use it. Add the following tests to `cost_value_test.go`:

```
func TestParseCostRange(t *testing.T) {
    is := is.New(t)
    var l meander.CostRange
    var err error
    l, err = meander.ParseCostRange("$$...$$$$")
    is.NoErr(err)
    is.Equal(l.From, meander.Cost2)
    is.Equal(l.To, meander.Cost3)
    l, err = meander.ParseCostRange("$...$$$$$")
    is.NoErr(err)
    is.Equal(l.From, meander.Cost1)
    is.Equal(l.To, meander.Cost5)
}
```

```
func TestCostRangeString(t *testing.T) {
    is := is.New(t)
    r := meander.CostRange{
        From: meander.Cost2,
        To:   meander.Cost4,
    }
    is.Equal("$$....$$$$", r.String())
}
```

We specify that passing in a string with two dollar characters first, followed by three dots and then three dollar characters should create a new `meander.CostRange` type that has `From` set to `meander.Cost2` and `To` set to `meander.Cost3`. We also use `is.NoErr` in order to assert that no error is returned when we parse our strings. The second test does the reverse by testing that the `CostRange.String` method, which returns the appropriate value.

To make our tests pass, add the following `CostRange` type and the associated `String` and `ParseString` functions:

```
type CostRange struct {
    From Cost
    To   Cost
}
func (r CostRange) String() string {
    return r.From.String() + "..." + r.To.String()
}
func ParseCostRange(s string) (CostRange, error) {
    var r CostRange
    segs := strings.Split(s, "...")
    if len(segs) != 2 {
        return r, errors.New("invalid cost range")
    }
    r.From = ParseCost(segs[0])
    r.To = ParseCost(segs[1])
    return r, nil
}
```

This allows us to convert a string such as `$....$$$$` to a structure that contains two `Cost` values: a `From` and `To` set and vice versa. If somebody passes in an invalid cost range (we just perform a simple check on the number of segments after splitting on the dots), then we return an error. You can do additional checking here if you want to, such as ensuring only dots and dollar signs are mentioned in the strings.

Querying the Google Places API

Now that we are capable of representing the results of the API, we need a way to represent and initiate the actual query. Add the following structure to `query.go`:

```
type Query struct {
    Lat          float64
    Lng          float64
    Journey      []string
    Radius       int
    CostRangeStr string
}
```

This structure contains all the information we will need in order to build up the query, all of which will actually come from the URL parameters in the requests from the client. Next, add the following `find` method, which will be responsible for making the actual request to Google's servers:

```
func (q *Query) find(types string) (*googleResponse, error) {
    u := "https://maps.googleapis.com/maps/api/place/nearbysearch/json"
    vals := make(url.Values)
    vals.Set("location", fmt.Sprintf("%g,%g", q.Lat, q.Lng))
    vals.Set("radius", fmt.Sprintf("%d", q.Radius))
    vals.Set("types", types)
    vals.Set("key", APIKey)
    if len(q.CostRangeStr) > 0 {
        r, err := ParseCostRange(q.CostRangeStr)
        if err != nil {
            return nil, err
        }
        vals.Set("minprice", fmt.Sprintf("%d", int(r.From)-1))
        vals.Set("maxprice", fmt.Sprintf("%d", int(r.To)-1))
    }
    res, err := http.Get(u + "?" + vals.Encode())
    if err != nil {
        return nil, err
    }
    defer res.Body.Close()
    var response googleResponse
    if err := json.NewDecoder(res.Body).Decode(&response); err != nil {
        return nil, err
    }
    return &response, nil
}
```

First, we build the request URL as per the Google Places API specification by appending the `url.Values` encoded string of the data for `lat`, `lng`, `radius`, and, of course, the `APIKey` values.



The `url.Values` type is actually a `map[string][]string` type, which is why we use `make` rather than `new`.

The `types` value we specify as an argument represents the kind of business to look for. If there is `CostRangeStr`, we parse it and set the `minprice` and `maxprice` values before finally calling `http.Get` to actually make the request. If the request is successful, we defer the closing of the response body and use a `json.Decoder` method to decode the JSON that comes back from the API into our `googleResponse` type.

Building recommendations

Next, we need to write a method that will allow us to make many calls to find for the different steps in a journey. Underneath the `find` method, add the following `Run` method to the `Query` struct:

```
// Run runs the query concurrently, and returns the results.
func (q *Query) Run() []interface{} {
    rand.Seed(time.Now().UnixNano())
    var w sync.WaitGroup
    var l sync.Mutex
    places := make([]interface{}, len(q.Journey))
    for i, r := range q.Journey {
        w.Add(1)
        go func(types string, i int) {
            defer w.Done()
            response, err := q.find(types)
            if err != nil {
                log.Println("Failed to find places:", err)
                return
            }
            if len(response.Results) == 0 {
                log.Println("No places found for", types)
                return
            }
            for _, result := range response.Results {
                for _, photo := range result.Photos {
                    photo.URL =
                        "https://maps.googleapis.com/maps/api/place/photo?" +
                        "maxwidth=400&maxheight=400&photoreference=" +
                        result.Photoreference + "&key=" + q.APIKey
                }
            }
        }(types, i)
    }
    w.Wait()
}
```

```
        "maxwidth=1000&photoreference=" + photo.PhotoRef + "&key="
        + APIKey
    }
}
randI := rand.Intn(len(response.Results))
l.Lock()
places[i] = response.Results[randI]
l.Unlock()
}(r, i)
}
w.Wait() // wait for everything to finish
return places
}
```

The first thing we do is set the random seed to the current time in nanoseconds since January 1, 1970 UTC. This ensures that every time we call the `Run` method and use the `rand` package, the results will be different. If we don't do this, our code would suggest the same recommendations every time, which defeats the object.

Since we need to make many requests to Google and since we want to make sure this is as quick as possible we are going to run all the queries at the same time by making concurrent calls to our `Query.find` method. So next, we create `sync.WaitGroup` and a map to hold the selected places along with a `sync.Mutex` method to allow many goroutines to safely access the map concurrently.

We then iterate over each item in the `Journey` slice, which might be `bar`, `cafe`, or `movie_theater`. For each item, we add 1 to the `WaitGroup` object and start a goroutine. Inside the routine, we first defer the `w.Done` call, informing the `WaitGroup` object that this request has completed before calling our `find` method to make the actual request.

Assuming no errors occurred and it was indeed able to find some places, we iterate over the results and build up a usable URL for any photos that might be present. According to the Google Places API, we are given a `photoreference` key, which we can use in another API call to get the actual image. To save our clients from having to have knowledge of the Google Places API at all, we build the complete URL for them.

We then lock the map locker and with a call to `rand.Intn`, pick one of the options at random and insert it into the right position in the `places` slice before unlocking `sync.Mutex`.

Finally, we wait for all goroutines to complete with a call to `w.Wait` before returning the places.

Handlers that use query parameters

Now we need to wire up our /recommendations call, so head back to main.go in the cmd/meander folder and add the following code inside the main function:

```
http.HandleFunc("/recommendations", cors(func(w
    http.ResponseWriter, r *http.Request) {
    q := &meander.Query{
        Journey: strings.Split(r.URL.Query().Get("journey"), "|"),
    }
    var err error
    q.Lat, err = strconv.ParseFloat(r.URL.Query().Get("lat"), 64)
    if err != nil {
        http.Error(w, err.Error(), http.StatusBadRequest)
        return
    }
    q.Lng, err = strconv.ParseFloat(r.URL.Query().Get("lng"), 64)
    if err != nil {
        http.Error(w, err.Error(), http.StatusBadRequest)
        return
    }
    q.Radius, err = strconv.Atoi(r.URL.Query().Get("radius"))
    if err != nil {
        http.Error(w, err.Error(), http.StatusBadRequest)
        return
    }
    q.CostRangeStr = r.URL.Query().Get("cost")
    places := q.Run()
    respond(w, r, places)
}))
```

This handler is responsible for preparing the `meander.Query` object and calling its `Run` method before responding with the results. The `http.Request` type's `URL` value exposes the `Query` data that provides a `Get` method which, in turn, looks up a value for a given key.

The `journey` string is translated from the `bar|cafe|movie_theater` format to a slice of strings by splitting on the pipe character. Then, a few calls to functions in the `strconv` package turn the string latitude, longitude, and radius values into numerical types. If the values are in an incorrect format, we will get an error, which we will then write out to the client using the `http.Error` helper with an `http.StatusBadRequest` status.

CORS

The final piece of the first version of our API will be to implement CORS, as we did in the previous chapter. See if you can solve this problem yourself before reading on about the solution in the next section.



If you are going to tackle this yourself, remember that your aim is to set the `Access-Control-Allow-Origin` response header to `*`. Also, consider the `http.HandlerFunc` wrapping we did in the previous chapter. The best place for this code is probably in the `cmd/meander` program, since that is what exposes the functionality through an HTTP endpoint.

In `main.go`, add the following `cors` function:

```
func cors(f http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Access-Control-Allow-Origin", "*")
        f(w, r)
    }
}
```

This familiar pattern takes in an `http.HandlerFunc` type and returns a new one that sets the appropriate header before calling the passed-in function. Now, we can modify our code to make sure that the `cors` function gets called for both of our endpoints. Update the appropriate lines in the `main` function:

```
func main() {
    meander.APIKey = "YOUR_API_KEY"
    http.HandleFunc("/journeys", cors(func(w http.ResponseWriter,
        r *http.Request) {
        respond(w, r, meander.Journeys)
    }))
    http.HandleFunc("/recommendations", cors(func(w http.ResponseWriter,
        r *http.Request) {
        q := &meander.Query{
            Journey: strings.Split(r.URL.Query().Get("journey"), "|"),
        }
        var err error
        q.Lat, err = strconv.ParseFloat(r.URL.Query().Get("lat"), 64)
        if err != nil {
            http.Error(w, err.Error(), http.StatusBadRequest)
            return
        }
        q.Lng, err = strconv.ParseFloat(r.URL.Query().Get("lng"), 64)
    }))
}
```

```
if err != nil {
    http.Error(w, err.Error(), http.StatusBadRequest)
    return
}
q.Radius, err = strconv.Atoi(r.URL.Query().Get("radius"))
if err != nil {
    http.Error(w, err.Error(), http.StatusBadRequest)
    return
}
q.CostRangeStr = r.URL.Query().Get("cost")
places := q.Run()
respond(w, r, places)
})
log.Println("serving meander API on :8080")
http.ListenAndServe(":8080", http.DefaultServeMux)
}
```

Now, calls to our API will be allowed from any domain without a cross-origin error occurring.



Can you see a way to smarten up the code by removing the multiple calls to `r.URL.Query()`? Perhaps do this once and cache the result in a local variable. Then, you can avoid parsing the query many times.

Testing our API

Now that we are ready to test our API, head to a console and navigate to the `cmd/meander` folder. Because our program imports the `meander` package, building the program will automatically build our `meander` package too.

Build and run the program:

```
go build -o meanderapi
./meanderapi
```

To see meaningful results from our API, let's take a minute to find your actual latitude and longitude. Head over to <http://mygeoposition.com/> and use the web tools to get the `x`, `y` values for a location you are familiar with.

Or, pick from these popular cities:

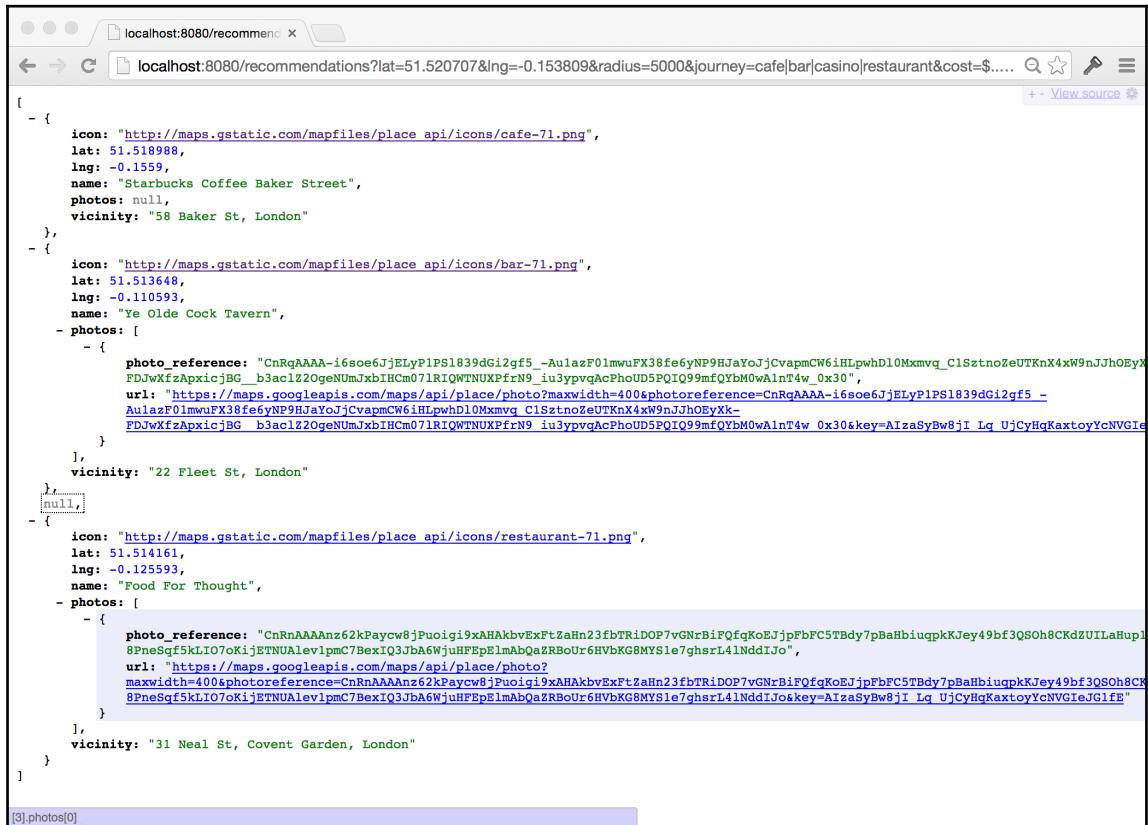
- London, England: 51.520707 x 0.153809
- New York, USA: 40.7127840 x -74.0059410

- Tokyo, Japan: 35.6894870 x 139.6917060
- San Francisco, USA: 37.7749290 x -122.4194160

Now, open a web browser and access the /recommendations endpoint with some appropriate values for the fields:

```
http://localhost:8080/recommendations?
lat=51.520707&lng=-0.153809&radius=5000&
journey=cafe|bar|casino|restaurant&
cost=$....$$$
```

The following screenshot shows what a sample recommendation around London might look like:



The screenshot shows a browser window with the URL `localhost:8080/recommendations?lat=51.520707&lng=-0.153809&radius=5000&journey=cafe|bar|casino|restaurant&cost=$....$$$`. The page displays a JSON object representing recommendations. The JSON structure includes an array of items, each containing details such as icon URL, coordinates, name, photos, and vicinity.

```
[
  {
    "icon": "http://maps.gstatic.com/mapfiles/place_api/icons/cafe-71.png",
    "lat": 51.518988,
    "lng": -0.1559,
    "name": "Starbucks Coffee Baker Street",
    "photos": null,
    "vicinity": "58 Baker St, London"
  },
  {
    "icon": "http://maps.gstatic.com/mapfiles/place_api/icons/bar-71.png",
    "lat": 51.513648,
    "lng": -0.110593,
    "name": "Ye Olde Cock Tavern",
    "photos": [
      {
        "photo_reference": "CnRgAAAA-i6soe6JjELyP1PSl839dGi2gf5_-AulazF0lmwuFX38fe6yNP9HJaYoJjCvapmCW6iHLpwhD10Mxmvg_C1SztnoZeUTKnX4xW9nJhOEYX-
        FDJwXfzApxicjBG__b3ac1z2OgeNUmJxbIHcm071RIQWTNUXPfrN9_iu3ypvqAcPhoud5PQiQ99mfQYbM0wA1nT4w_0x30",
        "url": "https://maps.googleapis.com/maps/api/place/photo?maxwidth=400&photoreference=CnRgAAAA-i6soe6JjELyP1PSl839dGi2gf5_-AulazF0lmwuFX38fe6yNP9HJaYoJjCvapmCW6iHLpwhD10Mxmvg_C1SztnoZeUTKnX4xW9nJhOEYX-
        FDJwXfzApxicjBG__b3ac1z2OgeNUmJxbIHcm071RIQWTNUXPfrN9_iu3ypvqAcPhoud5PQiQ99mfQYbM0wA1nT4w_0x30&key=A1zaSyBw8jI_Lq_UjCyHqKaxtoyYcNVGIE"
      }
    ],
    "vicinity": "22 Fleet St, London"
  },
  {
    "icon": "http://maps.gstatic.com/mapfiles/place_api/icons/restaurant-71.png",
    "lat": 51.514161,
    "lng": -0.125593,
    "name": "Food For Thought",
    "photos": [
      {
        "photo_reference": "CnRnAAAAnz62kPaycw8jPuoi9xAHAkbvExFtZaHn23fbTRiDOP7vGnrBiFQfqKoEJjpFbFC5TBdy7pBaHbiuqpkKJey49bf3QSoh8CKdZUILaHupI-
        8PneSgf5klIO7eKijETNUAllev1pmC7BexIQ3jbA6WjuHFFEpElmAbQaZRBouUr6HVbKG8MYS1e7ghsrL41NndiJo",
        "url": "https://maps.googleapis.com/maps/api/place/photo?
        maxwidth=400&photoreference=CnRnAAAAnz62kPaycw8jPuoi9xAHAkbvExFtZaHn23fbTRiDOP7vGnrBiFQfqKoEJjpFbFC5TBdy7pBaHbiuqpkKJey49bf3QSoh8CKdZUILaHupI-
        8PneSgf5klIO7eKijETNUAllev1pmC7BexIQ3jbA6WjuHFFEpElmAbQaZRBouUr6HVbKG8MYS1e7ghsrL41NndiJo&key=A1zaSyBw8jI_Lq_UjCyHqKaxtoyYcNVGIEJG1fE"
      }
    ],
    "vicinity": "31 Neal St, Covent Garden, London"
  }
]
```

[3].photos[0]

Feel free to play around with the values in the URL to see how powerful the simple API is by trying various journey strings, tweaking the locations, and trying different cost range value strings.

Web application

We are going to download a complete web application built to the same API specifications and point it at our implementation to see it come to life before our eyes. Head over to <https://github.com/matryer/goblueprints/tree/master/chapter7/meanderweb> download the `meanderweb` project into your `GOPATH` folder (alongside your root `meander` folder will do).

In a terminal, navigate to the `meanderweb` folder and build and run it:

```
go build -o meanderweb  
./meanderweb
```

This will start a website running on `localhost:8081`, which is hardcoded to look for the API running at `localhost:8080`. Because we added the CORS support, this won't be a problem despite them running on different domains.

Open a browser to `http://localhost:8081/` and interact with the application; while somebody else built the UI, it would be pretty useless without the API that we built in order to power it.

Summary

In this chapter, we built an API that consumes and abstracts the Google Places API to provide a fun and interesting way of letting users plan their days and evenings.

We started by writing some simple and short user stories that described what we wanted to achieve at a really high level without trying to design the implementation up front. In order to parallelize the project, we agreed upon the meeting point of the project as the API design, and we built toward it (as would our partners).

We embedded data directly in the code, avoiding the need to investigate, design, and implement a data store in the early stages of a project. By caring about how that data is accessed (via the API endpoint) instead, we allowed our future selves to completely change how and where the data is stored without breaking any apps that have been written with our API.

We implemented the `Facade` interface, which allows our structs and other types to provide public representations of them without revealing messy or sensitive details about our implementation.

Our foray into enumerators gave us a useful starting point to build enumerated types, even though there is no official support for them in the language. The `iota` keyword that we used lets us specify constants of our own numerical type, with incrementing values. The common `String` method that we implemented showed us how to make sure that our enumerated types don't become obscure numbers in our logs. At the same time, we also saw a real-world example of TDD and red/green programming, where we wrote unit tests that first fail but which we then go on to make pass by writing the implementation code.

In the next chapter, we are going to take a break from web services in order to build a backup tool for our code, where we'll explore how easy Go makes it for us to interact with the local filesystem.

8

Filesystem Backup

There are many solutions that provide filesystem backup capabilities. These include everything from apps such as Dropbox, Box, and Carbonite to hardware solutions such as Apple's Time Machine, Seagate, or network-attached storage products, to name a few. Most consumer tools provide some key automatic functionality, along with an app or website for you to manage your policies and content. Often, especially for developers, these tools don't quite do the things we need them to. However, thanks to Go's standard library (which includes packages such as `ioutil` and `os`), we have everything we need to build a backup solution that behaves exactly the way we need it to.

For our next project, we will build a simple filesystem backup for our source code projects that archive specified folders and save a snapshot of them every time we make a change. The change could be when we tweak a file and save it, when we add new files and folders, or even when we delete a file. We want to be able to go back to any point in time to retrieve old files.

Specifically, in this chapter, you will learn about the following topics:

- How to structure projects that consist of packages and command-line tools
- A pragmatic approach to persisting simple data across tool executions
- How the `os` package allows you to interact with a filesystem
- How to run code in an infinite timed loop while respecting *Ctrl + C*
- How to use `filepath.Walk` to iterate over files and folders
- How to quickly determine whether the contents of a directory have changed
- How to use the `archive/zip` package to zip files
- How to build tools that care about a combination of command-line flags and normal arguments

Solution design

We will start by listing some high-level acceptance criteria for our solution and the approach we want to take:

- The solution should create a snapshot of our files at regular intervals as we make changes to our source code projects
- We want to control the interval at which the directories are checked for changes
- Code projects are primarily text-based, so zipping the directories to generate archives will save a lot of space
- We will build this project quickly, while keeping a close watch over where we might want to make improvements later
- Any implementation decisions we make should be easily modified if we decide to change our implementation in the future
- We will build two command-line tools: the backend daemon that does the work and a user interaction utility that will let us list, add, and remove paths from the backup service

The project structure

It is common in Go solutions to have, in a single project, both a package that allows other Go programmers to use your capabilities and a command-line tool that allows end users to use your programs.

As we saw in the last chapter, a convention to structure such projects is emerging whereby we have the package in the main project project folder and the command-line tool inside a subfolder called `cmd` or `cmds` if you have multiple commands. Because all packages are equal in Go (regardless of the directory tree), you can import the package from the command subpackages, knowing you'll never need to import the commands from the project package (which is illegal as you can't have cyclical dependencies). This may seem like an unnecessary abstraction, but it is actually quite a common pattern and can be seen in the standard Go tool chain with examples such as `gofmt` and `goimports`.

For example, for our project, we are going to write a package called `backup` and two command-line tools: the daemon and the user interaction tool. We will structure our project in the following way:

```
/backup - package  
/backup/cmds/backup - user interaction tool  
/backup/cmds/backupd - worker daemon
```



The reason we don't just put code directly inside the `cmd` folder (even if we only had one command) is that when `go install` builds projects, it uses the name of the folder as the command name, and it wouldn't be very useful if all of our tools were called `cmd`.

The backup package

We are first going to write the `backup` package, of which we will become the first customer when we write the associated tools. The package will be responsible for deciding whether directories have changed and need backing up or not as well as actually performing the backup procedure.

Considering obvious interfaces first

One of the early things to think about when embarking on a new Go program is whether any interfaces stand out to you. We don't want to over-abstract or waste too much time upfront designing something that we know will change as we start to code, but that doesn't mean we shouldn't look for obvious concepts that are worth pulling out. If you're not sure, that is perfectly acceptable; you should write your code using concrete types and revisit potential abstractions after you have actually solved the problems.

However, since our code will archive files, the `Archiver` interface pops out as a candidate.

Create a new folder inside your `GOPATH/src` folder called `backup`, and add the following `archiver.go` code:

```
package backup
type Archiver interface {
    Archive(src, dest string) error
}
```

An `Archiver` interface will specify a method called `Archive`, which takes source and destination paths and returns an error. Implementations of this interface will be responsible for archiving the source folder and storing it in the destination path.



Defining an interface up front is a nice way to get some concepts out of our heads and into the code; it doesn't mean that this interface can't change as we evolve our solution as long as we remember the power of simple interfaces. Also, remember that most of the I/O interfaces in the `io` package expose only a single method.

From the very beginning, we have made the case that while we are going to implement ZIP files as our archive format, we could easily swap this out later with another kind of Archiver format.

Testing interfaces by implementing them

Now that we have the interface for our Archiver types, we are going to implement one that uses the ZIP file format.

Add the following struct definition to `archiver.go`:

```
type zipper struct{}
```

We are not going to export this type, which might make you jump to the conclusion that users outside of the package won't be able to make use of it. In fact, we are going to provide them with an instance of the type for them to use in order to save them from having to worry about creating and managing their own types.

Add the following exported implementation:

```
// Zip is an Archiver that zips and unzips files.  
var ZIP Archiver = (*zipper)(nil)
```

This curious snippet of Go voodoo is actually a very interesting way of exposing the intent to the compiler without using any memory (literally 0 bytes). We are defining a variable called `ZIP` of type `Archiver`, so from outside the package, it's pretty clear that we can use that variable wherever `Archiver` is needed if you want to zip things. Then, we assign it with `nil` cast to the type `*zipper`. We know that `nil` takes no memory, but since it's cast to a `zipper` pointer, and given that our `zipper` struct has no state, it's an appropriate way of solving a problem, which hides the complexity of code (and indeed the actual implementation) from outside users. There is no reason anybody outside of the package needs to know about our `zipper` type at all, which frees us up to change the internals without touching the externals at any time: the true power of interfaces.

Another handy side benefit to this trick is that the compiler will now be checking whether our `zipper` type properly implements the `Archiver` interface or not, so if you try to build this code, you'll get a compiler error:

```
./archiver.go:10: cannot use (*zipper)(nil) (type *zipper) as type  
Archiver in assignment:  
    *zipper does not implement Archiver (missing Archive method)
```

We see that our zipper type does not implement the Archive method as mandated in the interface.



You can also use the Archive method in test code to ensure that your types implement the interfaces they should. If you don't need to use the variable, you can always throw it away using an underscore and you'll still get the compiler help:

```
var _ Interface = (*Implementation)(nil)
```

To make the compiler happy, we are going to add the implementation of the Archive method for our zipper type.

Add the following code to archiver.go:

```
func (z *zipper) Archive(src, dest string) error {
    if err := os.MkdirAll(filepath.Dir(dest), 0777); err != nil {
        return err
    }
    out, err := os.Create(dest)
    if err != nil {
        return err
    }
    defer out.Close()
    w := zip.NewWriter(out)
    defer w.Close()
    return filepath.Walk(src, func(path string, info os.FileInfo, err error) error {
        if info.IsDir() {
            return nil // skip
        }
        if err != nil {
            return err
        }
        in, err := os.Open(path)
        if err != nil {
            return err
        }
        defer in.Close()
        f, err := w.Create(path)
        if err != nil {
            return err
        }
        _, err = io.Copy(f, in)
        if err != nil {
            return err
        }
    })
}
```

```
    }
    return nil
  })
}
```

You will also have to import the `archive/zip` package from the Go standard library. In our `Archive` method, we take the following steps to prepare writing to a ZIP file:

- Use `os.MkdirAll` to ensure that the destination directory exists. The `0777` code represents the file permissions with which you may need to create any missing directories
- Use `os.Create` to create a new file as specified by the `dest` path
- If the file is created without an error, defer the closing of the file with `defer out.Close()`
- Use `zip.NewWriter` to create a new `zip.Writer` type that will write to the file we just created and defer the closing of the writer

Once we have a `zip.Writer` type ready to go, we use the `filepath.Walk` function to iterate over the source directory, `src`.

The `filepath.Walk` function takes two arguments: the root path and a callback function to be called for every item (files and folders) it encounters while iterating over the filesystem.



Functions are first class types in Go, which means you can use them as argument types as well as global functions and methods. The `filepath.Walk` function specifies the second argument type as `filepath.WalkFunc`, which is a function with a specific signature. As long as we adhere to the signature (correct input and return arguments) we can write inline functions rather than worrying about the `filepath.WalkFunc` type at all. Taking a quick look at the Go source code tell us that the signature for `filepath.WalkFunc` matches the function we are passing in `func(path string, info os.FileInfo, err error) error`

The `filepath.Walk` function is recursive, so it will travel deep into subfolders too. The callback function itself takes three arguments: the full path of the file, the `os.FileInfo` object that describes the file or folder itself, and an error (it also returns an error in case something goes wrong). If any calls to the callback function result in an error (other than the special `SkipDir` error value) being returned, the operation will be aborted and `filepath.Walk` returns that error. We simply pass this up to the caller of `Archive` and let them worry about it, since there's nothing more we can do.

For each item in the tree, our code takes the following steps:

- If the `info.isdir` method tells us that the item is a folder, we just return `nil`, effectively skipping it. There is no reason to add folders to ZIP archives because the path of the files will encode that information for us.
- If an error is passed in (via the third argument), it means something went wrong when trying to access information about the file. This is uncommon, so we just return the error, which will eventually be passed out to the caller of `Archive`. As the implementor of `filepath.Walk`, you aren't forced to abort the operation here; you are free to do whatever makes sense in your individual case.
- Use `os.Open` to open the source file for reading, and if successful, defer its closing.
- Call `Create` on the `ZipWriter` object to indicate that we want to create a new compressed file and give it the full path of the file, which includes the directories it is nested inside.
- Use `io.Copy` to read all of the bytes from the source file and write them through the `ZipWriter` object to the ZIP file we opened earlier.
- Return `nil` to indicate no errors.

This chapter will not cover unit testing or **Test-driven Development (TDD)** practices, but feel free to write a test to ensure that our implementation does what it is meant to do.



Since we are writing a package, spend some time commenting on the exported pieces so far. You can use `golint` to help you find anything you may have missed.

Has the filesystem changed?

One of the biggest problems our backup system has is deciding whether a folder has changed or not in a cross-platform, predictable, and reliable way. After all, there's no point in creating a backup if nothing is different from the previous backup. A few things spring to mind when we think about this problem: should we just check the last modified date on the top-level folder? Should we use system notifications to be informed whenever a file we care about changes? There are problems with both of these approaches, and it turns out it's not a simple problem to solve.



Check out the `fsnotify` project at <https://fsnotify.org> (project source: <https://github.com/fsnotify>). The authors are attempting to build a cross-platform package for subscription to filesystem events. At the time of writing this, the project is still in its infancy and it's not a viable option for this chapter, but in the future, it could well become the standard solution for filesystem events.

We are, instead, going to generate an MD5 hash made up of all of the information that we care about when considering whether something has changed or not.

Looking at the `os.FileInfo` type, we can see that we can find out a lot of information about a file or folder:

```
type FileInfo interface {
    Name() string      // base name of the file
    Size() int64       // length in bytes for regular files;
                       // system-dependent for others
    Mode() FileMode   // file mode bits
    ModTime() time.Time // modification time
    IsDir() bool       // abbreviation for Mode().IsDir()
    Sys() interface{}  // underlying data source (can return nil)
}
```

To ensure we are aware of a variety of changes to any file in a folder, the hash will be made up of the filename and path (so if they rename a file, the hash will be different), size (if a file changes size, it's obviously different), the last modified date, whether the item is a file or folder, and the file mode bits. Even though we won't be archiving the folders, we still care about their names and the tree structure of the folder.

Create a new file called `dirhash.go` and add the following function:

```
package backup
import (
    "crypto/md5"
    "fmt"
    "io"
    "os"
    "path/filepath"
)
func DirHash(path string) (string, error) {
    hash := md5.New()
    err := filepath.Walk(path, func(path string, info os.FileInfo, err error) error {
        if err != nil {
            return err
        }
    })
    if err != nil {
        return "", err
    }
    return fmt.Sprintf("%x", hash.Sum(nil)), nil
}
```

```
    io.WriteString(hash, path)
    fmt.Fprintf(hash, "%v", info.IsDir())
    fmt.Fprintf(hash, "%v", info.ModTime())
    fmt.Fprintf(hash, "%v", info.Mode())
    fmt.Fprintf(hash, "%v", info.Name())
    fmt.Fprintf(hash, "%v", info.Size())
    return nil
})
if err != nil {
    return "", err
}
return fmt.Sprintf("%x", hash.Sum(nil)), nil
}
```

We first create a new `hash.Hash` function that knows how to calculate MD5s before using `filepath.Walk` again to iterate over all of the files and folders inside the specified path directory. For each item, assuming there are no errors, we write the differential information to the hash generator using `io.WriteString`, which lets us write a string to `io.Writer` and `fmt.Fprintf`, which does the same but exposes formatting capabilities at the same time, allowing us to generate the default value format for each item using the `%v` format verb.

Once each file has been processed, and assuming no errors occurred, we then use `fmt.Sprintf` to generate the result string. The `Sum` method in `hash.Hash` calculates the final hash value with the specified values appended. In our case, we do not want to append anything since we've already added all of the information we care about, so we just pass `nil`. The `%x` format verb indicates that we want the value to be represented in hex (base 16) with lowercase letters. This is the usual way of representing an MD5 hash.

Checking for changes and initiating a backup

Now that we have the ability to hash a folder and perform a backup, we are going to put the two together in a new type called `Monitor`. The `Monitor` type will have a map of paths with their associated hashes, a reference to any `Archiver` type (of course, we'll use `backup.ZIP` for now), and a destination string representing where to put the archives.

Create a new file called `monitor.go` and add the following definition:

```
type Monitor struct {
    Paths      map[string]string
    Archiver   Archiver
    Destination string
}
```

In order to trigger a check for changes, we are going to add the following `Now` method:

```
func (m *Monitor) Now() (int, error) {
    var counter int
    for path, lastHash := range m.Paths {
        newHash, err := DirHash(path)
        if err != nil {
            return counter, err
        }
        if newHash != lastHash {
            err := m.act(path)
            if err != nil {
                return counter, err
            }
            m.Paths[path] = newHash // update the hash
            counter++
        }
    }
    return counter, nil
}
```

The `Now` method iterates over every path in the map and generates the latest hash of that folder. If the hash does not match the hash from the map (generated the last time it checked), then it is considered to have changed and needs backing up again. We do this with a call to the as-yet-unwritten `act` method before then updating the hash in the map with this new hash.

To give our users a high-level indication of what happened when they called `Now`, we are also maintaining a counter, which we increment every time we back up a folder. We will use this later to keep our end users up to date on what the system is doing without bombarding them with information:

```
m.act undefined (type *Monitor has no field or method act)
```

The compiler is helping us again and reminding us that we have yet to add the `act` method:

```
func (m *Monitor) act(path string) error {
    dirname := filepath.Base(path)
    filename := fmt.Sprintf("%d.zip", time.Now().UnixNano())
    return m.ArchiverArchive(path, filepath.Join(m.Destination, dirname,
        filename))
}
```

Because we have done the heavy lifting in our ZIP Archiver type, all we have to do here is generate a filename, decide where the archive will go, and call the `Archive` method.



If the `Archive` method returns an error, the `act` method and then the `Now` method will each return it. This mechanism of passing errors up the chain is very common in Go and allows you to either handle cases where you can do something useful to recover or else defer the problem to somebody else.

The `act` method in the preceding code uses `time.Now().UnixNano()` to generate a timestamp filename and hardcodes the `.zip` extension.

Hardcoding is OK for a short while

Hardcoding the file extension like we have is OK in the beginning, but if you think about it, we have blended concerns a little here. If we change the `Archiver` implementation to use RAR or a compression format of our making, the `.zip` extension would no longer be appropriate.



Before reading on, think about what steps you might take to avoid this hardcoded. Where does the filename extension decision live? What changes would you need to make in order to avoid hardcoded?

The right place for the filename extensions decision is probably in the `Archiver` interface, since it knows the kind of archiving it will be doing. So we could add an `Ext()` string method and access that from our `act` method. But we can add a little extra power with not much extra work by allowing `Archiver` authors to specify the entire filename format rather than just the extension instead.

Back in `archiver.go`, update the `Archiver` interface definition:

```
type Archiver interface {
    DestFmt() string
    Archive(src, dest string) error
}
```

Our `zipper` type needs to now implement this:

```
func (z *zipper) DestFmt() string {
    return "%d.zip"
}
```

Now that we can ask our `act` method to get the whole format string from the `Archiver` interface, update the `act` method:

```
func (m *Monitor) act(path string) error {
```

```
    dirname := filepath.Base(path)
    filename := fmt.Sprintf(m.Archiver.DestFmt(), time.Now().UnixNano())
    return m.Archiver.Archive(path, filepath.Join(m.Destination, dirname,
        filename))
}
```

The user command-line tool

The first of two tools we will build allows the user to add, list, and remove paths for the backup daemon tool (which we will write later). You can expose a web interface or even use the binding packages for the desktop user interface integration, but we are going to keep things simple and build ourselves a command-line tool.

Create a new folder called `cmds` inside the `backup` folder and create another `backup` folder inside that so you have `backup/cmds/backup`.

Inside our new `backup` folder, add the following code to `main.go`:

```
func main() {
    var fatalErr error
    defer func() {
        if fatalErr != nil {
            flag.PrintDefaults()
            log.Fatalln(fatalErr)
        }
    }()
    var (
        dbpath = flag.String("db", "./backupdata", "path to database
directory")
    )
    flag.Parse()
    args := flag.Args()
    if len(args) < 1 {
        fatalErr = errors.New("invalid usage; must specify command")
        return
    }
}
```

We first define our `fatalErr` variable and defer the function that checks to ensure that value is `nil`. If it is not, it will print the error along with flag defaults and exit with a nonzero status code. We then define a flag called `db` that expects the path to the `filedb` database directory before parsing the flags and getting the remaining arguments and ensuring that there is at least one.

Persisting small data

In order to keep track of the paths and the hashes that we generate, we will need some kind of data storage mechanism that ideally works even when we stop and start our programs. We have lots of choices here: everything from a text file to a full horizontally scalable database solution. The Go ethos of simplicity tells us that building-in a database dependency to our little backup program would not be a great idea; rather, we should ask what the simplest way in which we can solve this problem is.

The `github.com/matryer/filedb` package is an experimental solution for just this kind of problem. It lets you interact with the filesystem as though it were a very simple, schemaless database. It takes its design lead from packages such as `mgo` and can be used in cases where data querying needs are very simple. In `filedb`, a database is a folder, and a collection is a file where each line represents a different record. Of course, this could all change as the `filedb` project evolves, but the interface, hopefully, won't.



Adding dependencies such as this to a Go project should be done very carefully because over time, dependencies go stale, change beyond their initial scope, or disappear altogether in some cases. While it sounds counterintuitive, you should consider whether copying and pasting a few files into your project is a better solution than relying on an external dependency. Alternatively, consider vendorizing the dependency by copying the entire package into the `vendor` folder of your command. This is akin to storing a snapshot of the dependency that you know works for your tool.

Add the following code to the end of the `main` function:

```
db, err := filedb.Dial(*dbpath)
if err != nil {
    fatalErr = err
    return
}
defer db.Close()
col, err := db.C("paths")
if err != nil {
    fatalErr = err
    return
}
```

Here, we use the `filedb.Dial` function to connect with the `filedb` database. In actuality, nothing much happens here except specifying where the database is, since there are no real database servers to connect to (although this might change in the future, which is why such provisions exist in the interface). If that was successful, we defer the closing of the database.

Closing the database does actually do something, since files may be open that need to be cleaned up.

Following the `mgo` pattern, next we specify a collection using the `C` method and keep a reference to it in the `col` variable. If an error occurs at any point, we assign it to the `fatalErr` variable and return.

To store data, we are going to define a type called `path`, which will store the full path and the last hash value and use JSON encoding to store this in our `filedb` database. Add the following struct definition above the `main` function:

```
type path struct {
    Path string
    Hash string
}
```

Parsing arguments

When we call `flag.Args` (as opposed to `os.Args`), we receive a slice of arguments excluding the flags. This allows us to mix flag arguments and non-flag arguments in the same tool.

We want our tool to be able to be used in the following ways:

- To add a path:

```
backup -db=/path/to/db add {path} [paths...]
```

- To remove a path:

```
backup -db=/path/to/db remove {path} [paths...]
```

- To list all paths:

```
backup -db=/path/to/db list
```

To achieve this, since we have already dealt with flags, we must check the first (non-flag) argument.

Add the following code to the `main` function:

```
switch strings.ToLower(args[0]) {
    case "list":
    case "add":
    case "remove":
}
```

Here, we simply switch on the first argument after setting it to lowercase (if the user types `backup LIST`, we still want it to work).

Listing the paths

To list the paths in the database, we are going to use a `ForEach` method on the path's `col` variable. Add the following code to the `list` case:

```
var path path
col.ForEach(func(i int, data []byte) bool {
    err := json.Unmarshal(data, &path)
    if err != nil {
        fatalErr = err
        return true
    }
    fmt.Printf("= %s\n", path)
    return false
})
```

We pass in a callback function to `ForEach`, which will be called for every item in that collection. We then unmarshal it from JSON into our `path` type, and just print it out using `fmt.Printf`. We return `false` as per the `filedb` interface, which tells us that returning `true` would stop iterating and that we want to make sure we list them all.

String representations for your own types

If you print structs in Go in this way, using the `%s` format verbs, you can get some messy results that are difficult for users to read. If, however, the type implements a `String()` string method, it will be used instead, and we can use this to control what gets printed. Below the `path` struct, add the following method:

```
func (p path) String() string {
    return fmt.Sprintf("%s [%s]", p.Path, p.Hash)
}
```

This tells the `path` type how it should represent itself as a string.

Adding paths

To add a path, or many paths, we are going to iterate over the remaining arguments and call the `InsertJSON` method for each one. Add the following code to the `add` case:

```
if len(args[1:]) == 0 {
    fatalErr = errors.New("must specify path to add")
    return
}
for _, p := range args[1:] {
    path := &path{Path: p, Hash: "Not yet archived"}
    if err := col.InsertJSON(path); err != nil {
        fatalErr = err
        return
    }
    fmt.Printf("+ %s\n", path)
}
```

If the user hasn't specified any additional arguments, for example if they just called `backup add` without typing any paths, we will return a fatal error. Otherwise, we do the work and print out the path string (prefixed with a `+` symbol) to indicate that it was successfully added. By default, we'll set the hash to the `Not yet archived` string literal this is an invalid hash but serves the dual purposes of letting the user know that it hasn't yet been archived as well as indicating as such to our code (given that a hash of the folder will never equal that string).

Removing paths

To remove a path, or many paths, we use the `RemoveEach` method for the path's collection. Add the following code to the `remove` case:

```
var path path
col.RemoveEach(func(i int, data []byte) (bool, bool) {
    err := json.Unmarshal(data, &path)
    if err != nil {
        fatalErr = err
        return false, true
    }
    for _, p := range args[1:] {
        if path.Path == p {
            fmt.Printf("- %s\n", path)
            return true, false
        }
    }
    return false, false
})
```

```
})
```

The callback function we provide to `RemoveEach` expects us to return two bool types: the first one indicates whether the item should be removed or not, and the second one indicates whether we should stop iterating or not.

Using our new tool

We have completed our simple backup command-line tool. Let's look at it in action. Create a folder called `backupdata` inside `backup/cmds/backup`; this will become the `filedb` database.

Build the tool in a terminal by navigating to the `main.go` file and running this:

```
go build -o backup
```

If all is well, we can now add a path:

```
./backup -db=./backupdata add ./test ./test2
```

You should see the expected output:

```
+ ./test [Not yet archived]  
+ ./test2 [Not yet archived]
```

Now let's add another path:

```
./backup -db=./backupdata add ./test3
```

You should now see the complete list:

```
./backup -db=./backupdata list
```

Our program should yield the following:

```
= ./test [Not yet archived]  
= ./test2 [Not yet archived]  
= ./test3 [Not yet archived]
```

Let's remove `test3` in order to make sure the `remove` functionality is working:

```
./backup -db=./backupdata remove ./test3
./backup -db=./backupdata list
```

This will take us back to this:

```
+ ./test [Not yet archived]
+ ./test2 [Not yet archived]
```

We are now able to interact with the `filedb` database in a way that makes sense for our use case. Next, we build the daemon program that will actually use our `backup` package to do the work.

The daemon backup tool

The `backup` tool, which we will call `backupd`, will be responsible for periodically checking the paths listed in the `filedb` database, hashing the folders to see whether anything has changed, and using the `backup` package to actually perform the archiving of the folders that need it.

Create a new folder called `backupd` alongside the `backup/cmds/backup` folder, and let's jump right into handling the fatal errors and flags:

```
func main() {
    var fatalErr error
    defer func() {
        if fatalErr != nil {
            log.Fatalln(fatalErr)
        }
    }()
    var (
        interval = flag.Duration("interval", 10 * time.Second, "interval
between
checks")
        archive  = flag.String("archive", "archive", "path to archive
location")
        dbpath   = flag.String("db", "./db", "path to filedb database")
    )
    flag.Parse()
}
```

You must be quite used to seeing this kind of code by now. We defer the handling of fatal errors before specifying three flags: `interval`, `archive`, and `db`. The `interval` flag represents the number of seconds between checks to see whether folders have changed, the `archive` flag is the path to the archive location where ZIP files will go, and the `db` flag is the path to the same `filedb` database that the `backup` command is interacting with. The usual call to `flag.Parse` sets the variables up and validates whether we're ready to move on.

In order to check the hashes of the folders, we are going to need an instance of `Monitor` that we wrote earlier. Append the following code to the `main` function:

```
m := &backup.Monitor{  
    Destination: *archive,  
    Archiver:     backup.ZIP,  
    Paths:        make(map[string]string),  
}
```

Here, we create `backup.Monitor` using the `archive` value as the `Destination` type. We'll use the `backup.ZIP` archiver and create a map ready for it to store the paths and hashes internally. At the start of the daemon, we want to load the paths from the database so that it doesn't archive unnecessarily as we stop and start things.

Add the following code to the `main` function:

```
db, err := filedb.Dial(*dbpath)  
if err != nil {  
    fatalErr = err  
    return  
}  
defer db.Close()  
col, err := db.C("paths")  
if err != nil {  
    fatalErr = err  
    return  
}
```

You have seen this code earlier too; it dials the database and creates an object that allows us to interact with the `paths` collection. If anything fails, we set `fatalErr` and return.

Duplicated structures

Since we're going to use the same path structure as we used in our user command-line tool program, we need to include a definition of it for this program too. Insert the following structure above the `main` function:

```
type path struct {
    Path string
    Hash string
}
```

The object-oriented programmers out there are no doubt screaming at the pages by now, demanding for this shared snippet to exist in one place only and not be duplicated in both programs. I urge you to resist this compulsion. These four lines of code hardly justify a new package, and therefore dependency for our code, when they can just as easily exist in both programs with very little overhead. Also, consider that we might want to add a `LastChecked` field to our `backupd` program so that we can add rules where each folder only gets archived once an hour at most. Our `backup` program doesn't care about this and will chug along perfectly happy with its view into what fields constitute a `path` structure.

Caching data

We can now query all existing paths and update the `Paths` map, which is a useful technique to increase the speed of a program, especially given slow or disconnected data stores. By loading the data into a cache (in our case, the `Paths` map), we can access it at lightning speed without having to consult the files each time we need information.

Add the following code to the body of the `main` function:

```
var path path
col.ForEach(func(_ int, data []byte) bool {
    if err := json.Unmarshal(data, &path); err != nil {
        fatalErr = err
        return true
    }
    m.Paths[path.Path] = path.Hash
    return false // carry on
})
if fatalErr != nil {
    return
}
if len(m.Paths) < 1 {
    fatalErr = errors.New("no paths - use backup tool to add at least one")
    return
}
```

}

Using the `ForEach` method again allows us to iterate over all the paths in the database. We unmarshal the JSON bytes into the same `path` structure as we used in our other program and set the values in the `Paths` map. Assuming that nothing goes wrong, we do a final check to make sure there is at least one path, and if not, we return with an error.



One limitation to our program is that it will not dynamically add paths once it has started. The daemon would need to be restarted. If this bothers you, you can always build in a mechanism that updates the `Paths` map periodically or uses some other kind of configuration management.

Infinite loops

The next thing we need to do is perform a check on the hashes right away to see whether anything needs archiving before entering into an infinite timed loop where we perform the check again at regular, specified intervals.

An infinite loop sounds like a bad idea; in fact, to some, it sounds like a bug. However, since we're talking about an infinite loop within this program, and since infinite loops can be easily broken with a simple `break` command, they're not as dramatic as they might sound. When we mix an infinite loop with a `select` statement that has no default case, we are able to run the code in a manageable way without gobbling up CPU cycles as we wait for something to happen. The execution will be blocked until one of the two channels receive data.

In Go, to write an infinite loop is as simple as running this:

```
for {}
```

The instructions inside the braces get executed over and over again, as quickly as the machine running the code can execute them. Again, this sounds like a bad plan unless you're careful about what you're asking it to do. In our case, we are immediately initiating a `select` case on the two channels that will block safely until one of the channels has something interesting to say.

Add the following code:

```
check(m, col)
signalChan := make(chan os.Signal, 1)
signal.Notify(signalChan, syscall.SIGINT, syscall.SIGTERM)
for {
    select {
```

```
    case <-time.After(*interval):
        check(m, col)
    case <-signalChan:
        // stop
        fmt.Println()
        log.Printf("Stopping...")
        return
    }
}
```

Of course, as responsible programmers, we care about what happens when the user terminates our programs. So after a call to the `check` method (which doesn't yet exist), we make a signal channel and use `signal.Notify` to ask for the termination signal to be given to the channel rather than it being handled automatically. In our infinite `for` loop, we select two possibilities: either the `timer` channel sends a message or the termination signal channel sends a message. If it's the `timer` channel message, we call `check` again; if it's `signalChan`, we go about terminating the program; otherwise, we'll loop back and wait.

The `time.After` function returns a channel that will send a signal (actually, the current time) after the specified time has elapsed. Since we are using `flag.Duration`, we can pass this (dereferenced via `*`) as the `time.Duration` argument directly into the function. Using `flag.Duration` also means that users can specify time durations in a human readable way, such as `10s` for 10 seconds or `1m` for a minute.

Finally, we return from the main function, causing the deferred statements to execute, such as closing the database connection.

Updating filedb records

All that is left is for us to implement the `check` function that should call the `Now` method on the `Monitor` type and update the database with new hashes if there are any.

Underneath the `main` function, add the following code:

```
func check(m *backup.Monitor, col *filedb.C) {
    log.Println("Checking...")
    counter, err := m.Now()
    if err != nil {
        log.Fatalln("failed to backup:", err)
    }
    if counter > 0 {
        log.Printf(" Archived %d directories\n", counter)
        // update hashes
        var path path
```

```
        col.SelectEach(func(_ int, data []byte) (bool, []byte, bool) {
            if err := json.Unmarshal(data, &path); err != nil {
                log.Println("failed to unmarshal data (skipping):", err)
                return true, data, false
            }
            path.Hash, _ = m.Paths[path.Path]
            newdata, err := json.Marshal(&path)
            if err != nil {
                log.Println("failed to marshal data (skipping):", err)
                return true, data, false
            }
            return true, newdata, false
        })
    } else {
        log.Println("  No changes")
    }
}
```

The `check` function first tells the user that a check is happening before immediately calling `Now`. If the `Monitor` type did any work for us, which is to ask whether it archived any files, we output them to the user and go on to update the database with the new values. The `SelectEach` method allows us to change each record in the collection if we so wish by returning the replacement bytes. So we unmarshal the bytes to get the path structure, update the hash value, and return the marshaled bytes. This ensures that the next time we start a `backupd` process, it will do so with the correct hash values.

Testing our solution

Let's see whether our two programs play nicely together. You may want to open two terminal windows for this, since we'll be running two programs.

We have already added some paths to the database, so let's use `backup` to see them:

```
./backup -db="./backupdata" list
```

You should see the two test folders; if you don't, refer to the *Adding paths* section:

```
= ./test [Not yet archived]  
= ./test2 [Not yet archived]
```

In another window, navigate to the `backupd` folder and create our two test folders, called `test` and `test2`.

Build `backupd` using the usual method:

```
go build -o backupd
```

Assuming all is well, we can now start the backup process, being sure to point the `db` path to the same path as we used for the `backup` program and specifying that we want to use a new folder called `archive` to store the ZIP files. For testing purposes, let's specify an interval of 5 seconds in order to save time:

```
./backupd -db="../backup/backupdata/" -archive=".archive" -  
interval=5s
```

Immediately, `backupd` should check the folders, calculate the hashes, note that they are different (to `Not yet archived`), and initiate the archive process for both folders. It will print the output that tells us this:

```
Checking...  
Archived 2 directories
```

Open the newly created `archive` folder inside `backup/cmds/backupd` and note that it has created two subfolders: `test` and `test2`. Inside these are compressed archive versions of the empty folders. Feel free to `unzip` one and see; nothing very exciting so far.

Meanwhile, back in the terminal window, `backupd` has been checking the folders for changes again:

```
Checking...  
No changes  
Checking...  
No changes
```

In your favorite text editor, create a new text file inside the `test2` folder, containing the word `test`, and save it as `one.txt`. After a few seconds, you will see that `backupd` has noticed the new file and created another snapshot inside the `archive/test2` folder.

Of course, it has a different filename because the time is different, but if you `unzip` it, you will notice that it has indeed created a compressed archive version of the folder.

Play around with the solution by taking the following actions:

- Change the contents of the `one.txt` file
- Add a file to the `test` folder too
- Delete a file

Summary

In this chapter, we successfully built a very simple backup system for your code projects. You can see how simple it would be to extend or modify the behavior of these programs. The scope for potential problems that you could go on to solve is limitless.

Rather than having a local archive destination folder like we did in the previous section, imagine mounting a network storage device and using that instead. Suddenly, you have off-site (or at least off-machine) backups of these vital files. You can easily set a Dropbox folder as the archive destination, which would mean that not only do you get access to the snapshots yourself, but a copy is also stored in the cloud and can even be shared with other users.

Extending the `Archiver` interface to support `Restore` operations (which would just use the `encoding/zip` package to unzip the files) allows you to build tools that can peer inside the archives and access the changes of individual files, much like Time Machine on a Mac allows you to do. Indexing the files gives you the complete search across the entire history of your code, much like GitHub does.

Since the filenames are timestamps, you could have backupd retiring old archives to less active storage mediums or summarized the changes into a daily dump.

Obviously, backup software exists, is well tested, and is used throughout the world, and it may be a smart move to focus on solving problems that haven't been solved yet. But when it requires such little effort to write small programs to get things done, it is often worth doing because of the control it gives you. When you write the code, you can get exactly what you want without compromise, and it's down to each individual to make that call.

Specifically, in this chapter, we explored how easy Go's standard library makes it to interact with the filesystem: opening files for reading, creating new files, and making directories. The `os` package mixed in with the powerful types from the `io` package, blended further with capabilities such as `encoding/zip` and others, gives a clear example of how extremely simple Go interfaces can be composed to deliver very powerful results.

9

Building a Q&A Application for Google App Engine

Google App Engine gives developers a **NoOps** (short for **No Operations**, indicating that developers and engineers have no work to do in order to have their code running and available) way of deploying their applications, and Go has been officially supported as a language option for some years now. Google's architecture runs some of the biggest applications in the world, such as Google Search, Google Maps, and Gmail, among others, so is a pretty safe bet when it comes to deploying our own code.

Google App Engine allows you to write a Go application, add a few special configuration files, and deploy it to Google's servers, where it will be hosted and made available in a highly available, scalable, and elastic environment. Instances will automatically spin up to meet demand and tear down gracefully when they are no longer needed with a healthy free quota and preapproved budgets.

Along with running application instances, Google App Engine makes available a myriad of useful services, such as fast and high-scale data stores, search, memcache, and task queues. Transparent load balancing means you don't need to build and maintain additional software or hardware to ensure servers don't get overloaded and that requests are fulfilled quickly.

In this chapter, we will build the API backend for a question and answer service similar to Stack Overflow or Quora and deploy it to Google App Engine. In the process, we'll explore techniques, patterns, and practices that can be applied to all such applications, as well as dive deep into some of the more useful services available to our application.

Specifically, in this chapter, you will learn:

- How to use the Google App Engine SDK for Go to build and test applications locally before deploying to the cloud
- How to use `app.yaml` to configure your application
- How Modules in Google App Engine let you independently manage the different components that make up your application
- How the Google Cloud Datastore lets you persist and query data at scale
- A sensible pattern for the modeling of data and working with keys in Google Cloud Datastore
- How to use the Google App Engine Users API to authenticate people with Google accounts
- A pattern to embed denormalized data into entities
- How to ensure data integrity and build counters using transactions
- Why maintaining a good line of sight in code helps improve maintainability
- How to achieve simple HTTP routing without adding a dependency to a third-party package

The Google App Engine SDK for Go

In order to run and deploy Google App Engine applications, we must download and configure the Go SDK. Head over to <https://cloud.google.com/appengine/downloads> and download the latest *Google App Engine SDK for Go* for your computer. The ZIP file contains a folder called `go_appengine`, which you should place in an appropriate folder outside of your `GOPATH`, for example, in `/Users/yourname/work/go_appengine`.



It is possible that the names of these SDKs will change in the future; if that happens, ensure that you consult the project home page for notes pointing you in the right direction at <https://github.com/matryer/gobuildprints>.

Next, you will need to add the `go_appengine` folder to your `$PATH` environment variable, much like what you did with the `go` folder when you first configured Go.

To test your installation, open a terminal and type this:

```
goapp version
```

You should see something like the following:

```
go version go1.6.1 (appengine-1.9.37) darwin/amd64
```



The actual version of Go is likely to differ and is often a few months behind actual Go releases. This is because the Cloud Platform team at Google needs to do work on its end to support new releases of Go.

The `goapp` command is a drop-in replacement for the `go` command with a few additional subcommands; so you can do things like `goapp test` and `goapp vet`, for example.

Creating your application

In order to deploy an application to Google's servers, we must use the Google Cloud Platform Console to set it up. In a browser, go to <https://console.cloud.google.com> and sign in with your Google account. Look for the **Create Project** menu item, which often gets moved around as the console changes from time to time. If you already have some projects, click on a project name to open a submenu, and you'll find it in there.



If you can't find what you're looking for, just search **Creating App Engine project** and you'll find it.

When the **New Project** dialog box opens, you will be asked for a name for your application. You are free to call it whatever you like (for example, Answers), but note the Project ID that is generated for you; you will need to refer to this when you configure your app later. You can also click on **Edit** and specify your own ID, but know that the value must be globally unique, so you'll have to get creative when thinking one up. In this book, we will use `answersapp` as the application ID, but you won't be able to use that one since it has already been taken.

You may need to wait a minute or two for your project to get created; there's no need to watch the page you can continue and check back later.

App Engine applications are Go packages

Now that the Google App Engine SDK for Go is configured and our application has been created, we can start building it.

In Google App Engine, an application is just a normal Go package with an `init` function that registers handlers via the `http.Handle` or `http.HandleFunc` functions. It does not need to be the `main` package like normal tools.

Create a new folder (somewhere inside your `GOPATH` folder) called `answersapp/api` and add the following `main.go` file:

```
package api
import (
    "io"
    "net/http"
)
func init() {
    http.HandleFunc("/", handleHello)
}
func handleHello(w http.ResponseWriter, r *http.Request) {
    io.WriteString(w, "Hello from App Engine")
}
```

You will be familiar with most of this by now, but note that there is no `ListenAndServe` call, and the handlers are set inside the `init` function rather than `main`. We are going to handle every request with our simple `handleHello` function, which will just write a welcoming string.

The `app.yaml` file

In order to turn our simple Go package into a Google App Engine application, we must add a special configuration file called `app.yaml`. The file will go at the root of the application or module, so create it inside the `answersapp/api` folder with the following contents:

```
application: YOUR_APPLICATION_ID_HERE
version: 1
runtime: go
api_version: go1
handlers:
- url: /.*
  script: _go_app
```

The file is a simple human-(and machine) readable configuration file in **YAML (Yet Another Markup Language)** format refer to yaml.org for more details). The following table describes each property:

Property	Description
application	The application ID (copied and pasted from when you created your project).
version	Your application version number you can deploy multiple versions and even split traffic between them to test new features, among other things. We'll just stick with version 1 for now.
runtime	The name of the runtime that will execute your application. Since this is a Go book and since we're building a Go application, we'll use go.
api_version	The <code>go1</code> api version is the runtime version supported by Google; you can imagine that this could be <code>go2</code> in the future.
handlers	A selection of configured URL mappings. In our case, everything will be mapped to the special <code>_go_app</code> script, but you can also specify static files and folders here.

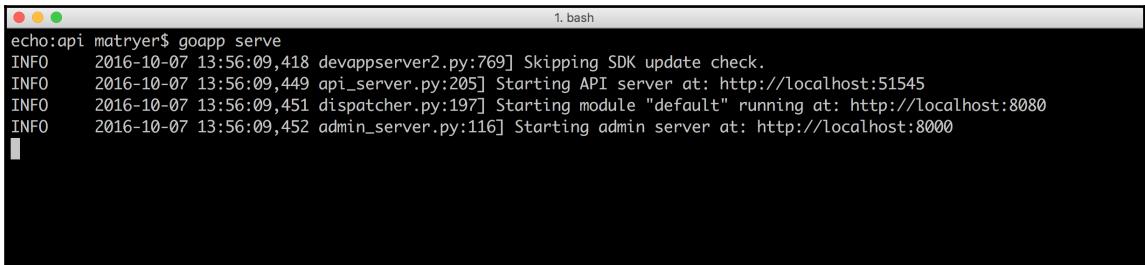
Running simple applications locally

Before we deploy our application, it makes sense to test it locally. We can do this using the App Engine SDK we downloaded earlier.

Navigate to your `answersapp/api` folder and run the following command in a terminal:

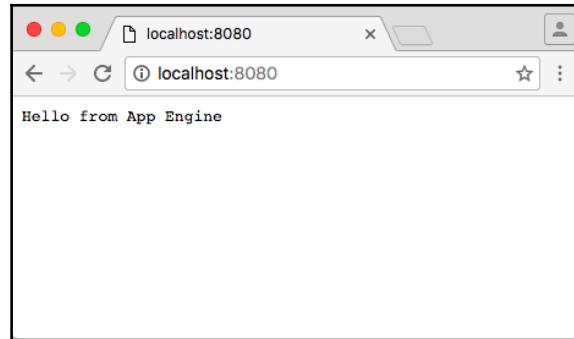
```
goapp serve
```

You should see the following output:



```
echo:api matryer$ goapp serve
INFO    2016-10-07 13:56:09,418 devappserver2.py:769] Skipping SDK update check.
INFO    2016-10-07 13:56:09,449 api_server.py:205] Starting API server at: http://localhost:51545
INFO    2016-10-07 13:56:09,451 dispatcher.py:197] Starting module "default" running at: http://localhost:8080
INFO    2016-10-07 13:56:09,452 admin_server.py:116] Starting admin server at: http://localhost:8000
```

This indicates that an API server is running locally on port :56443, an admin server is running on :8000, and our application (the module `default`) is now serving at `localhost:8080`, so let's hit that one in a browser.



As you can see by the `Hello` from App Engine response, our application is running locally. Navigate to the admin server by changing the port from :8080 to :8000.

The screenshot shows the Google App Engine Development SDK 1.9.37 web interface. The title bar says "Instances". The URL in the address bar is "localhost:8000/instances". The main content area has a header "Google App Engine" and "Development SDK 1.9.37". Below this, it says "dev~theanswersapp". On the left, there's a sidebar with links: Instances, Datastore Viewer, Datastore Indexes, Datastore Stats, Interactive Console, Memcache Viewer, Blobstore Viewer, Task Queues, Cron Jobs, XMPP, Inbound Mail, and Full Text Search. The main right-hand section is titled "Instances" and shows a table for the "default" instance group. The table has columns: Latency (ms), QPS, Total Requests, and Runtime. There are two rows of data:

	Latency (ms)	QPS	Total Requests	Runtime
badb1ad78b3bea224c934d7374eea37aef1a	0.0	0.00	0	go
dbdcbad42b38a1e6e1e5ec721fc7628df1	0.0	0.00	4	

The preceding screenshot shows the web portal that we can use to interrogate the internals of our application, including viewing running instances, inspecting the data store, managing task queues, and more.

Deploying simple applications to Google App Engine

To truly understand the power of Google App Engine's NoOps promise, we are going to deploy this simple application to the cloud. Back in the terminal, stop the server by hitting `Ctrl+C` and run the following command:

```
goapp deploy
```

Your application will be packaged and uploaded to Google's servers. Once it's finished, you should see something like the following:

```
Completed update of app: theanswersapp, version: 1
```

It really is as simple as that.

You can prove this by navigating to the endpoint you get for free with every Google App Engine application, remembering to replace the application ID with your own: https://YOUR_APPLICATION_ID_HERE.appspot.com/.

You will see the same output as earlier (the font may render differently since Google's servers will make assumptions about the content type that the local dev server doesn't).



The application is being served over HTTP/2 and is already capable of pretty massive scale, and all we did was write a `config` file and a few lines of code.

Modules in Google App Engine

A module is a Go package that can be versioned, updated, and managed independently. An app might have a single module, or it can be made up of many modules, each distinct but part of the same application with access to the same data and services. An application must have a default module even if it doesn't do much.

Our application will be made up of the following modules:

Description	The module name
The obligatory default module	default
An API package delivering RESTful JSON	api
A static website serving HTML, CSS, and JavaScript that makes AJAX calls to the API module	web

Each module will be a Go package and will, therefore, live inside its own folder.

Let's reorganize our project into modules by creating a new folder alongside the `api` folder called `default`.

We are not going to make our default module do anything other than use it for configuration, as we want our other modules to do all the meaningful work. But if we leave this folder empty, the Google App Engine SDK will complain that it has nothing to build.

Inside the `default` folder, add the following placeholder `main.go` file:

```
package defaultmodule
func init() {}
```

This file does nothing except allow our `default` module to exist



It would have been nice for our package names to match the folders, but `default` is a reserved keyword in Go, so we have a good reason to break that rule.

The other module in our application will be called `web`, so create another folder alongside the `api` and `default` folders called `web`. In this chapter, we are only going to build the API for our application and cheat by downloading the `web` module.

Head over to the project home page at <https://github.com/matryer/goblueprints>, access the content for **Second Edition**, and look for the download link for the *web components for Chapter 9, Building a Q&A Application for Google App Engine* in the Downloads section of the README file. The ZIP file contains the source files for the `web` component, which should be unzipped and placed inside the `web` folder.

Now, our application structure should look like this:

```
/answersapp/api
/answersapp/default
/answersapp/web
```

Specifying modules

To specify which module our `api` package will become, we must add a property to the `app.yaml` inside our `api` folder. Update it to include the `module` property:

```
application: YOUR_APPLICATION_ID_HERE
version: 1
runtime: go
module: api
api_version: go1
handlers:
- url: /.*
  script: _go_app
```

Since our default module will need to be deployed as well, we also need to add an `app.yaml` configuration file to it. Duplicate the `api/app.yaml` file inside `default/app.yaml`, changing the module to `default`:

```
application: YOUR_APPLICATION_ID_HERE
version: 1
runtime: go
module: default
api_version: go1
handlers:
- url: /.*
  script: _go_app
```

Routing to modules with `dispatch.yaml`

In order to route traffic appropriately to our modules, we will create another configuration file called `dispatch.yaml`, which will let us map URL patterns to the modules.

We want all traffic beginning with the `/api/` path to be routed to the `api` module and everything else to the `web` module. As mentioned earlier, we won't expect our `default` module to handle any traffic, but it will have more utility later.

In the `answersapp` folder (alongside our module folders not inside any of the module folders), create a new file called `dispatch.yaml` with the following contents:

```
application: YOUR_APPLICATION_ID_HERE
dispatch:
- url: "*/api/*"
  module: api
- url: "*/**"
  module: web
```

The same `application` property tells the Google App Engine SDK for Go which application we are referring to, and the `dispatch` section routes URLs to modules.

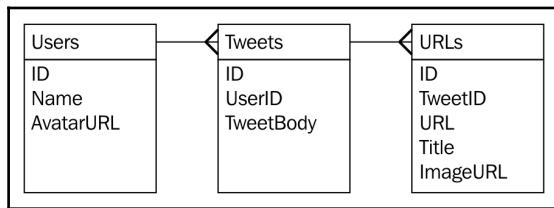
Google Cloud Datastore

One of the services available to App Engine developers is Google Cloud Datastore, a NoSQL document database built for automatic scaling and high performance. Its limited featureset guarantees very high scale, but understanding the caveats and best practices is vital to a successful project.

Denormalizing data

Developers with experience of relational databases (RDBMS) will often aim to reduce data redundancy (trying to have each piece of data appear only once in their database) by **normalizing** data, spreading it across many tables, and adding references (foreign keys) before joining it back via a query to build a complete picture. In schemaless and NoSQL databases, we tend to do the opposite. We **denormalize** data so that each document contains the complete picture it needs, making read times extremely fast since it only needs to go and get a single thing.

For example, consider how we might model tweets in a relational database such as MySQL or Postgres:

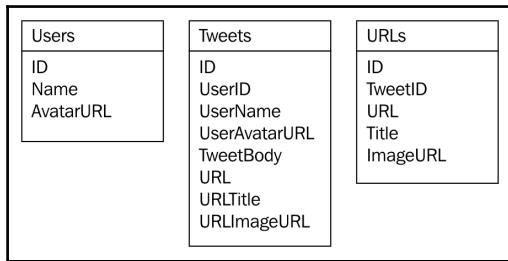


A tweet itself contains only its unique ID, a foreign key reference to the Users table representing the author of the tweet, and perhaps many URLs that were mentioned in TweetBody.

One nice feature of this design is that a user can change their Name or AvatarURL and it will be reflected in all of their tweets, past and future, something you wouldn't get for free in a denormalized world.

However, in order to present a tweet to the user, we must load the tweet itself, look up (via a join) the user to get their name and avatar URL, and then load the associated data from the URLs table in order to show a preview of any links. At scale, this becomes difficult because all three tables of data might well be physically separated from each other, which means lots of things need to happen in order to build up this complete picture.

Consider what a denormalized design would look like instead:



We still have the same three buckets of data, except that now our tweet contains everything it needs in order to render to the user without having to look up data from anywhere else. The hardcore relational database designers out there are realizing what this means by now, and it is no doubt making them feel uneasy.

Following this approach means that:

- Data is repeated – AvatarURL in User is repeated as UserAvatarURL in the tweet (waste of space, right?)
- If the user changes their AvatarURL, UserAvatarURL in the tweet will be out of date

Database design, at the end of the day, comes down to physics. We are deciding that our tweet is going to be read far more times than it is going to be written, so we'd rather take the pain upfront and take a hit in storage. There's nothing wrong with repeated data as long as there is an understanding about which set is the master set and which is duplicated for speed.

Changing data is an interesting topic in itself, but let's think about a few reasons why we might be OK with the trade-offs.

Firstly, the speed benefit to reading tweets is probably worth the unexpected behavior of changes to master data not being reflected in historical documents; it would be perfectly acceptable to decide to live with this emerged functionality for that reason.

Secondly, we might decide that it makes sense to keep a snapshot of data at a specific moment in time. For example, imagine if someone tweets asking whether people like their profile picture. If the picture changed, the tweet context would be lost. For a more serious example, consider what might happen if you were pointing to a row in an Addresses table for an order delivery and the address later changed. Suddenly, the order might look like it was shipped to a different place.

Finally, storage is becoming increasingly cheaper, so the need for normalizing data to save space is lessened. Twitter even goes as far as copying the entire tweet document for each of your followers. 100 followers on Twitter means that your tweet will be copied at least 100 times, maybe more for redundancy. This sounds like madness to relational database enthusiasts, but Twitter is making smart trade-offs based on its user experience; they'll happily spend a lot of time writing a tweet and storing it many times to ensure that when you refresh your feed, you don't have to wait very long to get updates.



If you want to get a sense of the scale of this, check out the Twitter API and look at what a tweet document consists of. It's a lot of data. Then, go and look at how many followers Lady Gaga has. This has become known in some circles as "the Lady Gaga problem" and is addressed by a variety of different technologies and techniques that are out of the scope of this chapter.

Now that we have an understanding of good NoSQL design practices, let's implement the types, functions, and methods required to drive the data part of our API.

Entities and data access

To persist data in Google Cloud Datastore, we need a struct to represent each entity. These entity structures will be serialized and deserialized when we save and load data through the `datastore` API. We can add helper methods to perform the interactions with the data store, which is a nice way to keep such functionality physically close to the entities themselves. For example, we will model an answer with a struct called `Answer` and add a `Create` method that in turn calls the appropriate function from the `datastore` package. This prevents us from bloating our HTTP handlers with lots of data access code and allows us to keep them clean and simple instead.

One of the foundation blocks of our application is the concept of a question. A question can be asked by a user and answered by many. It will have a unique ID so that it is addressable (referable in a URL), and we'll store a timestamp of when it was created.

Create a new file inside `answersapp` called `questions.go` and add the following struct function:

```
type Question struct {
    Key *datastore.Key `json:"id" datastore:"-"`
    CTime time.Time `json:"created"`
    Question string `json:"question"`
    User UserCard `json:"user"`
    AnswersCount int `json:"answers_count"`
```

}

The structure describes a question in our application. Most of it will seem quite obvious, as we've done similar things in the previous chapters. The `UserCard` struct represents a denormalized `User` entity, both of which we'll add later.



You can import the `datastore` package in your Go project using this:

```
import "google.golang.org/appengine/datastore"
```

It's worth spending a little time understanding the `datastore.Key` type.

Keys in Google Cloud Datastore

Every entity in Datastore has a key, which uniquely identifies it. They can be made up of either a string or an integer depending on what makes sense for your case. You are free to decide the keys for yourself or let Datastore automatically assign them for you; again, your use case will usually decide which is the best approach to take and we'll explore both in this chapter.

Keys are created using the `datastore.NewKey` and `datastore.NewIncompleteKey` functions and are used to put and get data into and out of Datastore via the `datastore.Get` and `datastore.Put` functions.

In Datastore, keys and entity bodies are distinct, unlike in MongoDB or SQL technologies, where it is just another field in the document or record. This is why we are excluding `Key` from our `Question` struct with the `datastore:"-"` field tag. Like the `json` tags, this indicates that we want Datastore to ignore the `Key` field altogether when it is getting and putting data.

Keys may optionally have parents, which is a nice way of grouping associated data together and Datastore makes certain assurances about such groups of entities, which you can read more about in the Google Cloud Datastore documentation online.

Putting data into Google Cloud Datastore

Before we save data into Datastore, we want to ensure that our question is valid. Add the following method underneath the Question struct definition:

```
func (q Question) OK() error {
    if len(q.Question) < 10 {
        return errors.New("question is too short")
    }
    return nil
}
```

The `OK` function will return an error if something is wrong with the question, or else it will return `nil`. In this case, we just check to make sure the question has at least 10 characters.

To persist this data in the data store, we are going to add a method to the `Question` struct itself. At the bottom of `questions.go`, add the following code:

```
func (q *Question) Create(ctx context.Context) error {
    log.Debugf(ctx, "Saving question: %s", q.Question)
    if q.Key == nil {
        q.Key = datastore.NewIncompleteKey(ctx, "Question", nil)
    }
    user, err := UserFromAEUser(ctx)
    if err != nil {
        return err
    }
    q.User = user.Card()
    q.CTime = time.Now()
    q.Key, err = datastore.Put(ctx, q.Key, q)
    if err != nil {
        return err
    }
    return nil
}
```

The `Create` method takes a pointer to `Question` as the receiver, which is important because we want to make changes to the fields.



If the receiver was `(q Question)` without `*`, we would get a copy of the question rather than a pointer to it, and any changes we made to it would only affect our local copy and not the original `Question` struct itself.

The first thing we do is use `log` (from the <https://godoc.org/google.golang.org/appengine/log> package) to write a debug statement saying we are saving the question. When you run your code in a development environment, you will see this appear in the terminal; in production, it goes into a dedicated logging service provided by Google Cloud Platform.

If the key is `nil` (that means this is a new question), we assign an incomplete key to the field, which informs Datastore that we want it to generate a key for us. The three arguments we pass are `context.Context` (which we must pass to all datastore functions and methods), a string describing the kind of entity, and the parent key; in our case, this is `nil`.

Once we know there is a key in place, we call a method (which we will add later) to get or create `User` from an App Engine user and set it to the question and then set the `CTime` field (created time) to `time.Now`, timestamping the point at which the question was asked.

Once we have our `Question` function in good shape, we call `datastore.Put` to actually place it inside the data store. As usual, the first argument is `context.Context`, followed by the question key and the question entity itself.

Since Google Cloud Datastore treats keys as separate and distinct from entities, we have to do a little extra work if we want to keep them together in our own code. The `datastore.Put` method returns two arguments: the complete key and `error`. The key argument is actually useful because we're sending in an incomplete key and asking the data store to create one for us, which it does during the put operation. If successful, it returns a new `datastore.Key` object to us, representing the completed key, which we then store in our `Key` field in the `Question` object.

If all is well, we return `nil`.

Add another helper to update an existing question:

```
func (q *Question) Update(ctx context.Context) error {
    if q.Key == nil {
        q.Key = datastore.NewIncompleteKey(ctx, "Question", nil)
    }
    var err error
    q.Key, err = datastore.Put(ctx, q.Key, q)
    if err != nil {
        return err
    }
    return nil
}
```

This method is very similar except that it doesn't set the `CTime` or `User` fields, as they will already have been set.

Reading data from Google Cloud Datastore

Reading data is as simple as putting it with the `datastore.Get` method, but since we want to maintain keys in our entities (and `datastore` methods don't work like that), it's common to add a helper function like the one we are going to add to `questions.go`:

```
func GetQuestion(ctx context.Context, key *datastore.Key)  
(*Question, error) {  
    var q Question  
    err := datastore.Get(ctx, key, &q)  
    if err != nil {  
        return nil, err  
    }  
    q.Key = key  
    return &q, nil  
}
```

The `GetQuestion` function takes `context.Context` and the `datastore.Key` method of the question to get. It then does the simple task of calling `datastore.Get` and assigning the key to the entity before returning it. Of course, errors are handled in the usual way.

This is a nice pattern to follow so that users of your code know that they never have to interact with `datastore.Get` and `datastore.Put` directly but rather use the helpers that can ensure the entities are properly populated with the keys (along with any other tweaks that they might want to do before saving or after loading).

Google App Engine users

Another service we are going to make use of is the Google App Engine Users API, which provides the authentication of Google accounts (and Google Apps accounts).

Create a new file called `users.go` and add the following code:

```
type User struct {  
    Key *datastore.Key `json:"id" datastore:"-"  
    UserID string `json:"-"  
    DisplayName string `json:"display_name"  
    AvatarURL string `json:"avatar_url"  
    Score int `json:"score"  
}
```

Similar to the `Question` struct, we have `Key` and a few fields that make up the `User` entity. This struct represents an object that belongs to our application that describes a user; we will have one for every authenticated user in our system, but this isn't the same user object that we'll get from the Users API.

Importing the <https://godoc.org/google.golang.org/appengine/user> package and calling the `user.Current(context.Context)` function will return either `nil` (if no user is authenticated) or a `user.User` object. This object belongs to the Users API and isn't suitable for our data store, so we need to write a helper function that will translate the App Engine user into our `User`.



Watch out that `goimports` doesn't automatically import `os/user` instead; sometimes it's best if you handle imports manually.

Add the following code to `users.go`:

```
func UserFromAEUser(ctx context.Context) (*User, error) {
    aeuser := user.Current(ctx)
    if aeuser == nil {
        return nil, errors.New("not logged in")
    }
    var appUser User
    appUser.Key = datastore.NewKey(ctx, "User", aeuser.ID, 0, nil)
    err := datastore.Get(ctx, appUser.Key, &appUser)
    if err != nil && err != datastore.ErrNoSuchEntity {
        return nil, err
    }
    if err == nil {
        return &appUser, nil
    }
    appUser.UserID = aeuser.ID
    appUser.DisplayName = aeuser.String()
    appUser.AvatarURL = gravatarURL(aeuser.Email)
    log.Infof(ctx, "saving new user: %s", aeuser.String())
    appUser.Key, err = datastore.Put(ctx, appUser.Key, &appUser)
    if err != nil {
        return nil, err
    }
    return &appUser, nil
}
```

We get the currently authenticated user by calling `user.Current`, and if it is `nil`, we return with an error. This means that the user is not logged in and the operation cannot complete. Our web package will be checking and ensuring that users are logged in for us, so by the time they hit an API endpoint, we'll expect them to be authenticated.

We then create a new `appUser` variable (which is of our `User` type) and set `datastore.Key`. This time, we aren't making an incomplete key; instead, we are using `datastore.NewKey` and specifying a string ID, matching the User API ID. This key predictability means that not only will there only be one `User` entity per authenticated user in our application, but it also allows us to load a `User` entity without having to use a query.



If we had the App Engine User ID as a field instead, we would need to do a query to find the record we are interested in. Querying is a more expensive operation compared to a direct `Get` method, so this approach is always preferred if you can do it.

We then call `datastore.Get` to attempt to load the `User` entity. If this is the first time the user has logged in, there will be no entity and the returned error will be the special `datastore.ErrNoSuchEntity` variable. If that's the case, we set the appropriate fields and use `datastore.Put` to save it. Otherwise, we just return the loaded `User`.



Note that we are checking for early returns in this function. This is to ensure that it is easy to read the execution flow of our code without having to follow it in and out of indented blocks. I call this the line of sight of code and have written about it on my blog at <https://medium.com/@matryer>.

For now, we'll use Gravatar again for avatar pictures, so add the following helper function to the bottom of `users.go`:

```
func gravatarURL(email string) string {
    m := md5.New()
    io.WriteString(m, strings.ToLower(email))
    return fmt.Sprintf("//www.gravatar.com/avatar/%x", m.Sum(nil))
}
```

Embedding denormalized data

If you recall, our `Question` type doesn't take the author as `User`; rather, the type was `UserCard`. When we embed denormalized data into other entities, sometimes we will want them to look slightly different from the master entity. In our case, since we do not store the key in the `User` entity (remember the `Key` fields have `datastore:"-"`), we need to have a new type that stores the key.

At the bottom of `users.go`, add the `UserCard` struct and the associated helper method for `User`:

```
type UserCard struct {
    Key          *datastore.Key `json:"id"`
    DisplayName string        `json:"display_name"`
    AvatarURL   string        `json:"avatar_url"`
}
func (u User) Card() UserCard {
    return UserCard{
        Key:      u.Key,
        DisplayName: u.DisplayName,
        AvatarURL: u.AvatarURL,
    }
}
```

Note that `UserCard` doesn't specify a `datastore` tag, so the `Key` field will indeed be persisted in the data store. Our `Card()` helper function just builds and returns `UserCard` by copying the values of each field. This seems wasteful but offers great control, especially if you want embedded data to look very different from its original entity.

Transactions in Google Cloud Datastore

Transactions allow you to specify a series of changes to the data store and commit them as one. If any of the individual operations fails, the whole transaction will not be applied. This is extremely useful if you want to maintain counters or have multiple entities that depend on each other's state. During a transaction in Google Cloud Datastore, all entities that are read are locked (other code is prevented from making changes) until the transaction is complete, providing an additional sense of security and preventing data races.



If you were building a bank (it seems crazy, but the guys at Monzo in London are indeed building a bank using Go), you might represent user accounts as an entity called `Account`. To transfer money from one account to another, you'd need to make sure the money was deducted from account A and deposited into account B as a single transaction. If either fails, people aren't going to be happy (to be fair, if the deduction operation failed, the owner of account A would probably be happy because B would get the money without it costing A anything).

To see where we are going to use transactions, let's first add model answers to the questions.

Create a new file called `answers.go` and add the following struct and validation method:

```
type Answer struct {
    Key      *datastore.Key `json:"id" datastore:"-"`
    Answer   string          `json:"answer"`
    CTime    time.Time       `json:"created"`
    User     UserCard        `json:"user"`
    Score    int              `json:"score"`
}
func (a Answer) OK() error {
    if len(a.Answer) < 10 {
        return errors.New("answer is too short")
    }
    return nil
}
```

`Answer` is similar to a question, has `datastore.Key` (which will not be persisted), has `CTime` to capture the timestamp, and embeds `UserCard` (representing the person answering the question). It also has a `Score` integer field, which will go up and down as users vote on the answers.

Using transactions to maintain counters

Our `Question` struct has a field called `AnswerCount`, where we intend to store an integer that represents the number of answers that a question has solicited.

First, let's look at what can happen if we don't use a transaction to keep track of the `AnswerCount` field by tracking the concurrent activity of answers 4 and 5 of a question:

Step	Answer 4	Answer 5	Question.AnswerCount
1	Load question	Load question	3
2	AnswerCount=3	AnswerCount=3	3
3	AnswerCount++	AnswerCount++	3
4	AnswerCount=4	AnswerCount=4	3
5	Save the answer and question	Save the answer and question	4

You can see from the table that without locking Question, AnswerCount would end up being 4 instead of 5 if the answers came in at the same time. Locking with a transaction will look more like this:

Step	Answer 4	Answer 5	Question.AnswerCount
1	Lock the question	Lock the question	3
2	AnswerCount=3	Waiting for unlock	3
3	AnswerCount++	Waiting for unlock	3
4	Save the answer and question	Waiting for unlock	4
5	Release lock	Waiting for unlock	4
6	Finished	Lock the question	4
7		AnswerCount=4	4
8		AnswerCount++	4
9		Save the answer and question	5

In this case, whichever answer obtains the lock first will perform its operation, and the other operation will wait before continuing. This is likely to slow down the operation (since it has to wait for the other one to finish), but that's a price worth paying in order to get the numbers right.



It's best to keep the amount of work inside a transaction as small as possible because you are essentially blocking other people while the transaction is underway. Outside of transactions, Google Cloud Datastore is extremely fast because it isn't making the same kinds of guarantees.

In code, we use the `datastore.RunInTransaction` function. Add the following to `answers.go`:

```
func (a *Answer) Create(ctx context.Context, questionKey *datastore.Key) error {
    a.Key = datastore.NewIncompleteKey(ctx, "Answer", questionKey)
    user, err := UserFromAEUser(ctx)
    if err != nil {
        return err
    }
    a.User = user.Card()
    a.CTime = time.Now()
    err = datastore.RunInTransaction(ctx, func(ctx context.Context) error {
        q, err := GetQuestion(ctx, questionKey)
        if err != nil {
            return err
        }
        a.Question = q
        _, err = datastore.Put(ctx, a.Key, a)
        return err
    })
    if err != nil {
        return err
    }
}
```

```
        return err
    }
    err = a.Put(ctx)
    if err != nil {
        return err
    }
    q.AnswersCount++
    err = q.Update(ctx)
    if err != nil {
        return err
    }
    return nil
}, &datastore.TransactionOptions{XG: true})
if err != nil {
    return err
}
return nil
}
```

We first create a new incomplete key (using the `Answer` kind) and set the parent as the question key. This will mean that the question will become the ancestor to all these answers.



Ancestor keys are special in Google Cloud Datastore, and it is recommended that you read about the nuances behind them in the documentation on the Google Cloud Platform website.

Using our `UserFromAEUser` function, we get the user who is answering the question and set `UserCard` inside `Answer` before setting `CTime` to the current time, as done earlier.

Then, we start our transaction by calling the `datastore.RunInTransaction` function that takes a context as well as a function where the transactional code will go. There is a third argument, which is a set of `datastore.TransactionOptions` that we need to use in order to set `XG` to `true`, which informs the data store that we'll be performing a transaction across entity groups (both `Answer` and `Question` kinds).



When it comes to writing your own functions and designing your own APIs, it is highly recommended that you place any function arguments at the end; otherwise, inline function blocks such as the ones in the preceding code obscure the fact that there is another argument afterwards. It's quite difficult to realize that the `TransactionOptions` object is an argument being passed into the `RunInTransaction` function, and I suspect somebody on the Google team regrets this decision.

Transactions work by providing a new context for us to use, which means that code inside the transaction function looks the same, as if it weren't in a transaction. This is a nice piece of API design (and it means that we can forgive the function for not being the final argument).

Inside the transaction function, we use our `GetQuestion` helper to load the question. Loading data inside the transaction function is what obtains a lock on it. We then put the answer to save it, update the `AnswerCount` integer, and update the question. If all is well (provided none of these steps returns an error), the answer will be saved and `AnswerCount` will increase by one.

If we do return an error from our transaction function, the other operations are canceled and the error is returned. If that happens, we'll just return that error from our `Answer.Create` method and let the user try again.

Next, we are going to add our `GetAnswer` helper, which is similar to our `GetQuestion` function:

```
func GetAnswer(ctx context.Context, answerKey *datastore.Key)
(*Answer, error) {
    var answer Answer
    err := datastore.Get(ctx, answerKey, &answer)
    if err != nil {
        return nil, err
    }
    answer.Key = answerKey
    return &answer, nil
}
```

Now we are going to add our `Put` helper method in `answers.go`:

```
func (a *Answer) Put(ctx context.Context) error {
    var err error
    a.Key, err = datastore.Put(ctx, a.Key, a)
    if err != nil {
        return err
    }
    return nil
}
```

These two functions are very similar to the `GetQuestion` and `Question.Put` methods, but let's resist the temptation of abstracting it and drying up the code for now.

Avoiding early abstraction

Copying and pasting is generally seen by programmers as a bad thing because it is usually possible to abstract the general idea and **DRY (Don't repeat yourself)** up the code.

However, it is worth resisting the temptation to do this right away because it is very easy to design a bad abstraction, which you are then stuck with since your code will start to depend on it. It is better to duplicate the code in a few places first and later revisit them to see whether a sensible abstraction is lurking there.

Querying in Google Cloud Datastore

So far, we have only been putting and getting single objects into and out of Google Cloud Datastore. When we display a list of answers to a question, we want to load all of these answers in a single operation, which we can do with `datastore.Query`.

The querying interface is a fluent API, where each method returns the same object or a modified object, allowing you to chain calls together. You can use it to build up a query consisting of ordering, limits, ancestors, filters, and so on. We will use it to write a function that will load all the answers for a given question, showing the most popular (those with a higher `Score` value) first.

Add the following function to `answers.go`:

```
func GetAnswers(ctx context.Context, questionKey *datastore.Key)
 ([]*Answer, error) {
    var answers []*Answer
    answerKeys, err := datastore.NewQuery("Answer").
        Ancestor(questionKey).
        Order("-Score").
        Order("-CTime").
        GetAll(ctx, &answers)
    for i, answer := range answers {
        answer.Key = answerKeys[i]
    }
    if err != nil {
        return nil, err
    }
    return answers, nil
}
```

We first create an empty slice of pointers to `Answer` and use `datastore.NewQuery` to start building a query. The `Ancestor` method indicates that we're looking only for answers that belong to the specific question, where the `Order` method calls specify that we want to first order by descending `Score` and then by the newest first. The `GetAll` method performs the operation, which takes in a pointer to our slice (where the results will go) and returns a new slice containing all the keys.



The order of the keys returned will match the order of the entities in the slice. This is how we know which key corresponds to each item.

Since we are keeping keys and the entity fields together, we range over the answers and assign `answer.Key` to the corresponding `datastore.Key` argument returned from `GetAll`.



We are keeping our API simple for the first version by not implementing paging, but ideally you would need to; otherwise, as the number of questions and answers grows, you will end up trying to deliver everything in a single request, which would overwhelm the user and maybe the servers.

If we had a step in our application of authorizing the answer (to protect it from spam or inappropriate content), we might want to add an additional filter for `Authorized` to be true, in which case we could do this:

```
datastore.NewQuery("Answer").  
  Filter("Authorized =", true)
```



For more information on querying and filtering, consult the Google Cloud Datastore API documentation online.

Another place where we need to query data is when we show the top questions on the home page of our app. Our first version of top questions will just show those questions that have the most answers; we consider them to be the most interesting, but you could change this functionality in the future without breaking the API to order by score or even question views.

We will build `Query` on the `Question` kind and use the `Order` method to first order by the number of answers (with the highest first), followed by time (also, highest/latest first). We will also use the `Limit` method to make sure we only select the top 25 questions for this API. Later, if we implement paging, we can even make this dynamic.

In `questions.go`, add the `TopQuestions` function:

```
func TopQuestions(ctx context.Context) ([]*Question, error) {
    var questions []*Question
    questionKeys, err := datastore.NewQuery("Question").
        Order("-AnswersCount").
        Order("-CTime").
        Limit(25).
        GetAll(ctx, &questions)
    if err != nil {
        return nil, err
    }
    for i := range questions {
        questions[i].Key = questionKeys[i]
    }
    return questions, nil
}
```

This code is similar to loading the answers, and we end up returning a slice of `Question` objects or an error.

Votes

Now that we have modeled questions and answers in our application, it's time to think about how voting might work.

Let's design it a little:

- Users vote answers up and down based on their opinion of them
- Answers are ordered by their score so the best ones appear first
- Each person is allowed one vote per answer
- If a user votes again, they should replace their previous vote

We will make use of a few things we have learned so far in this chapter; transactions will help us ensure the correct score is calculated for answers, and we'll use predictable keys again to ensure that each person gets only one vote per answer.

We will first build a structure to represent each vote and use field tags to be a little more specific about how we want the data store to index our data.

Indexing

Reads from Google Cloud Datastore are extremely fast due to the extensive use of indexes. By default, every field in our structure is indexed. Queries that attempt to filter on fields that aren't indexed will fail (the method will return an error); the data store doesn't fall back to scanning like some other technologies do because it's considered too slow. If one query filters two or more fields, an additional index must be added that is composed of all fields.

A structure with 10 fields would perform multiple write operations when you put it: one for the entity itself and one for each index that needs to be updated. So it is sensible to turn off indexing for fields that are you not planning to query on.

In `questions.go`, add the `datastore` field tags to the `Question` structure:

```
type Question struct {
    Key *datastore.Key `json:"id" datastore:"-"`
    CTime time.Time `json:"created" datastore:",noindex"`
    Question string `json:"question" datastore:",noindex"`
    User UserCard `json:"user"`
    AnswersCount int `json:"answers_count"`
}
```

The addition of the `datastore:",noindex"` field tags will tell the data store not to index these fields.



The `, noindex` value beginning with a comma is a little confusing. The value is essentially a list of comma-separated arguments, the first being the name we want the data store to use when storing each field (just like it does for the `json` tag). Since we don't want to say anything about the name we want the data store to use the real field name we are omitting it; so the first argument is empty, and the second argument is `noindex`.

Do this for fields that we do not want indexed in the Answer structure:

```
type Answer struct {
    Key *datastore.Key `json:"id" datastore:"-"`
    Answer string `json:"answer" datastore:",noindex"`
    CTime time.Time `json:"created"`
    User UserCard `json:"user" datastore:",noindex"`
    Score int `json:"score"`
}
```

And for the Vote structure, do this:

```
type Vote struct {
    Key *datastore.Key `json:"id" datastore:"-"`
    MTime time.Time `json:"last_modified" datastore:",noindex"`
    Question QuestionCard `json:"question" datastore:",noindex"`
    Answer AnswerCard `json:"answer" datastore:",noindex"`
    User UserCard `json:"user" datastore:",noindex"`
    Score int `json:"score" datastore:",noindex"`
}
```

You can also add a noindex declaration to all fields inside our card types: AnswerCard, UserCard, and QuestionCard.



The fields we have left without noindex will be used in queries, and we need to make sure Google Cloud Datastore does indeed maintain indexes on these fields.

Embedding a different view of entities

Now it's time to create our Vote structure, which we'll do inside a new file called votes.go:

```
type Vote struct {
    Key *datastore.Key `json:"id" datastore:"-"`
    MTime time.Time `json:"last_modified" datastore:",noindex"`
    Question QuestionCard `json:"question" datastore:",noindex"`
    Answer AnswerCard `json:"answer" datastore:",noindex"`
    User UserCard `json:"user" datastore:",noindex"`
    Score int `json:"score" datastore:",noindex"`
}
```

A `Vote` structure contains many of our embeddable card types representing `Question`, `Answer` and `User` casting the vote. It also contains a `Score` integer, which will be either 1 or -1 (depending on whether they voted up or down). We will also keep track of when they cast their vote (or last changed it) with the `MTimetime.Time` field.



You can use pointers to the `*Card` types in the `Vote` struct if you like. This would save additional copies being made when if you pass the `Vote` object in and out of functions, but that would mean that any changes made inside these functions would affect the original data rather than just their local copy. In most situations, there isn't much of a performance benefit to using pointers and it might be considered simpler to omit them. This book deliberately mixes both approaches to show you how they work, but you should understand the implications before making a decision.

Like our `UserCard` method, we are going to add appropriate versions for questions and answers, but this time we are going to be more selective about which fields should be included and which should be left out.

In `questions.go`, add the `QuestionCard` type and the associated helper method:

```
type QuestionCard struct {
    Key *datastore.Key `json:"id" datastore:",noindex"`
    Question string `json:"question" datastore:",noindex"`
    User     UserCard `json:"user" datastore:",noindex"`
}

func (q Question) Card() QuestionCard {
    return QuestionCard{
        Key:      q.Key,
        Question: q.Question,
        User:     q.User,
    }
}
```

The `QuestionCard` type captures the `Question` string and who asked it (our `UserCard` method, again), but we are leaving out the `CTime` and `AnswersCount` fields.

Let's add `AnswerCard` to `answers.go` next:

```
type AnswerCard struct {
    Key      *datastore.Key `json:"id" datastore:",noindex"`
    Answer   string          `json:"answer" datastore:",noindex"`
    User     UserCard        `json:"user" datastore:",noindex"`
}

func (a Answer) Card() AnswerCard {
```

```
    return AnswerCard{
        Key:      a.Key,
        Answer:   a.Answer,
        User:     a.User,
    }
}
```

Similarly, we are only capturing the `Answer` string and `User` and excluding `CTime` and `Score`.

Deciding which fields to capture and which to omit is entirely dependent on the user experience you wish to provide. We might decide that when we show a vote, we want to show the score of `Answer` at the time, or we might want to show the current score of `Answer` regardless of what it was at the time the vote was cast. Perhaps we want to send a push notification to the user who wrote the answer saying something like "Blanca has up-voted your answer to Ernesto's question it now has a score of 15", in which case we would need to grab the `Score` field too.

Casting a vote

Before our API is a complete feature, we need to add the ability for users to cast votes. We'll break this piece into two functions in order to increase the readability of our code.

Inside `votes.go`, add the following function:

```
func CastVote(ctx context.Context, answerKey *datastore.Key, score int)
(*Vote, error) {
    question, err := GetQuestion(ctx, answerKey.Parent())
    if err != nil {
        return nil, err
    }
    user, err := UserFromAEUser(ctx)
    if err != nil {
        return nil, err
    }
    var vote Vote
    err = datastore.RunInTransaction(ctx, func(ctx context.Context) error {
        var err error
        vote, err = castVoteInTransaction(ctx, answerKey, question, user,
            score)
        if err != nil {
            return err
        }
    })
    return nil
```

```
}, &datastore.TransactionOptions{XG: true})
if err != nil {
    return nil, err
}
return &vote, nil
}
```

The CastVote function takes (along with the obligatory `Context`) `datastore.Key` for the answer that is being voted for and a score integer. It loads the question and the current user, starts a data store transaction, and passes execution off to the `castVoteInTransaction` function.

Accessing parents via `datastore.Key`

Our `CastVote` function could require that we know `datastore.Key` for `Question` so that we can load it. But one nice feature about ancestor keys is that from the key alone, you can access the parent key. This is because the hierarchy of keys is maintained in the key itself, a bit like a path.

Three answers to question 1 might have these keys:

- `Question,1/Answer,1`
- `Question,1/Answer,2`
- `Question,1/Answer,3`

The actual details of how keys work under the hood are kept internal to the `datastore` package and could change at any time. So it is smart to only rely on things that the API guarantees such as being able to access the parent via the `Parent` method.

Line of sight in code

The cost of writing a function is relatively low compared to the cost of maintaining it, especially in successful, long-running projects. So it is worth taking the time to ensure the code is readable by our future selves and others.

Code can be said to have a good line of sight if it is easy to glance at and if it understands the usual, expected flow of the statements (the happy path). In Go, we can achieve this by following a few simple rules when we write code:

- Align the happy path to the left edge so that you can scan down a single column and see the expected flow of execution
- Don't hide the happy path logic inside a nest of indented braces
- Exit early from your function
- Indent only to handle errors or edge cases
- Extract functions and methods to keep bodies small and readable



There are a few more details to writing good line of sight code, which are outlined and maintained at <http://bit.ly/lineofsightincode>.

In order to prevent our `CastVote` function from becoming too big and difficult to follow, we have broken out the core functionality into its own function, which we will now add to `votes.go`:

```
func castVoteInTransaction(ctx context.Context, answerKey *datastore.Key,
    question *Question, user *User, score int) (Vote, error) {
    var vote Vote
    answer, err := GetAnswer(ctx, answerKey)
    if err != nil {
        return vote, err
    }
    voteKeyStr := fmt.Sprintf("%s:%s", answerKey.Encode(), user.Key.Encode())
    voteKey := datastore.NewKey(ctx, "Vote", voteKeyStr, 0, nil)
    var delta int // delta describes the change to answer score
    err = datastore.Get(ctx, voteKey, &vote)
    if err != nil && err != datastore.ErrNoSuchEntity {
        return vote, err
    }
    if err == datastore.ErrNoSuchEntity {
        vote = Vote{
            Key:      voteKey,
            User:     user.Card(),
            Answer:   answer.Card(),
            Question: question.Card(),
            Score:    score,
        }
    } else {
        // they have already voted - so we will be changing
        // this vote
    }
}
```

```
        delta = vote.Score * -1
    }
    delta += score
    answer.Score += delta
    err = answer.Put(ctx)
    if err != nil {
        return vote, err
    }
    vote.Key = voteKey
    vote.Score = score
    vote.MTime = time.Now()
    err = vote.Put(ctx)
    if err != nil {
        return vote, err
    }
    return vote, nil
}
```

While this function is long, its line of sight isn't too bad. The happy path flows down the left edge, and we only indent to return early in case of errors and the case where we create a new `Vote` object. This means that we can easily track what it is doing.

We take in the answer key, the related question, the user casting the vote and the score, and return a `Vote` object, or else an error if something goes wrong.

First, we get the answer which, since we're inside a transaction, will lock it until the transaction is complete (or stops due to an error).

We then build the key for this vote, which is made up of the keys of both the answer and the user encoded into a single string. This means that only one `Vote` entity will exist in the data store for each user/answer pair; so a user may only have one vote per answer as per our design.

We then use the vote key to attempt to load the `Vote` entity from the data store. Of course, the first time a user votes on a question, no entity will exist, which we can check by seeing whether the error returned from `datastore.Get` is the special `datastore.ErrNoSuchEntity` value or not. If it is, we create the new `Vote` object, setting the appropriate fields.

We are maintaining a score `delta` integer, which will represent the number that needs to be added to the answer score after the vote has happened. When it's the first time a user has voted on a question, the delta will be either 1 or -1. If they are changing their vote from down to up (-1 to 1), the delta will be 2, which cancels out the previous vote and adds the new one. We multiply the delta by -1 to undo the previous vote if there was one (if `err != datastore.ErrNoSuchEntity`). This has the nice effect of also not making any difference (`delta` will be 0) if they happen to cast the same vote twice in either direction.

Finally, we change the score on the answer and put it back into the data store before updating the final fields in our `Vote` object and putting that in too. We then return and our `CastVote` function exits the `datastore.RunInTransaction` function block, thus releasing `Answer` and letting others cast their votes on it too.

Exposing data operations over HTTP

Now that we have built all of our entities and the data access methods that operate on them, it's time to wire them up to an HTTP API. This will feel more familiar as we have already done this kind of thing a few times in the book.

Optional features with type assertions

When you use interface types in Go, you can perform type assertions to see whether the objects implement other interfaces, and since you can write interfaces inline, it is possible to very easily find out whether an object implements a specific function.

If `v` is `interface{}`, we can see whether it has the `OK` method using this pattern:

```
if obj, ok := v.(interface{ OK() error }); ok {
    // v has OK() method
} else {
    // v does not have OK() method
}
```

If the `v` object implements the method described in the interface, `ok` will be `true` and `obj` will be an object on which the `OK` method can be called. Otherwise, `ok` will be `false`.



One problem with this approach is that it hides the secret functionality from users of the code, so you must either document the function very well in order to make it clear or perhaps promote the method to its own first-class interface and insist that all objects implement it. Remember that we must always seek clear code over clever code. As a side exercise, see whether you can add the interface and use it in the decode signature instead.

We are going to add a function that will help us decode JSON request bodies and, optionally, validate the input. Create a new file called `http.go` and add the following code:

```
func decode(r *http.Request, v interface{}) error {
    err := json.NewDecoder(r.Body).Decode(v)
    if err != nil {
        return err
    }
    if valid, ok := v.(interface {
        OK() error
    }); ok {
        err = valid.OK()
        if err != nil {
            return err
        }
    }
    return nil
}
```

The decode function takes `http.Request` and a destination value called `v`, which is where the data from the JSON will go. We check whether the `OK` method is implemented, and if it is, we call it. We expect `OK` to return `nil` if the object looks good; otherwise, we expect it to return an error that explains what is wrong. If we get an error, we'll return it and let the calling code deal with it.

If all is well, we return `nil` at the bottom of the function.

Response helpers

We are going to add a pair of helper functions that will make responding to API requests easy. Add the `respond` function to `http.go`:

```
func respond(ctx context.Context, w http.ResponseWriter,
    r *http.Request, v interface{}, code int) {
    var buf bytes.Buffer
    err := json.NewEncoder(&buf).Encode(v)
```

```
if err != nil {
    respondErr(ctx, w, r, err, http.StatusInternalServerError)
    return
}
w.Header().Set("Content-Type",
    "application/json; charset=utf-8")
w.WriteHeader(code)
_, err = buf.WriteTo(w)
if err != nil {
    log.Errorf(ctx, "respond: %s", err)
}
}
```

The `respond` method contains a `context`, `ResponseWriter`, `Request`, the object to respond with, and a status code. It encodes `v` into an internal buffer before setting the appropriate headers and writing the response.

We are using a buffer here because it's possible that the encoding might fail. If it does so but has already started writing the response, the 200 OK header will be sent to the client, which is misleading. Instead, encoding to a buffer lets us be sure that completes without issue before deciding what status code to respond with.

Now add the `respondErr` function at the bottom of `http.go`:

```
func respondErr(ctx context.Context, w http.ResponseWriter,
    r *http.Request, err error, code int) {
    errObj := struct {
        Error string `json:"error"`
    }{Error: err.Error()}
    w.Header().Set("Content-Type", "application/json; charset=utf-8")
    w.WriteHeader(code)
    err = json.NewEncoder(w).Encode(errObj)
    if err != nil {
        log.Errorf(ctx, "respondErr: %s", err)
    }
}
```

This function writes `error` wrapped in a struct that embeds the error string as a field called `error`.

Parsing path parameters

Some of our API endpoints will need to pull IDs out of the path string, but we don't want to add any dependencies to our project (such as an external router package); instead, we are going to write a simple function that will parse path parameters for us.

Let's first write a test that will explain how we want our path parsing to work. Create a file called `http_test.go` and add the following unit test:

```
func TestPathParams(t *testing.T) {
    r, err := http.NewRequest("GET", "1/2/3/4/5", nil)
    if err != nil {
        t.Errorf("NewRequest: %s", err)
    }
    params := pathParams(r, "one/two/three/four")
    if len(params) != 4 {
        t.Errorf("expected 4 params but got %d: %v", len(params), params)
    }
    for k, v := range map[string]string{
        "one":    "1",
        "two":    "2",
        "three":  "3",
        "four":   "4",
    } {
        if params[k] != v {
            t.Errorf("%s: %s != %s", k, params[k], v)
        }
    }
    params = pathParams(r, "one/two/three/four/five/six")
    if len(params) != 5 {
        t.Errorf("expected 5 params but got %d: %v", len(params), params)
    }
    for k, v := range map[string]string{
        "one":    "1",
        "two":    "2",
        "three":  "3",
        "four":   "4",
        "five":   "5",
    } {
        if params[k] != v {
            t.Errorf("%s: %s != %s", k, params[k], v)
        }
    }
}
```

We expect to be able to pass in a pattern and have a map returned that discovers the values from the path in `http.Request`.

Run the test (with `go test -v`) and note that it fails.

At the bottom of `http.go`, add the following implementation to make the test pass:

```
func pathParams(r *http.Request, pattern string) map[string]string{
    params := map[string]string{}
    pathSegs := strings.Split(strings.Trim(r.URL.Path, "/"), "/")
    for i, seg := range strings.Split(strings.Trim(pattern, "/"), "/") {
        if i > len(pathSegs)-1 {
            return params
        }
        params[seg] = pathSegs[i]
    }
    return params
}
```

The function breaks the path from the specific `http.Request` and builds a map of the values with keys taken from breaking the pattern path. So for a pattern of `/questions/{id}` and a path of `/questions/123`, it would return the following map:

```
questions: questions
id:          123
```

Of course, we'd ignore the `questions` key, but `id` will be useful.

Exposing functionality via an HTTP API

Now we have all the tools we need in order to put together our API: helper functions to encode and decode data payloads in JSON, path parsing functions, and all the entities and data access functionality to persist and query data in Google Cloud Datastore.

HTTP routing in Go

The three endpoints we are going to add in order to handle questions are outlined in the following table:

HTTP request	Description
POST /questions	Ask a new question
GET /questions/{id}	Get the question with the specific ID
GET /questions	Get the top questions

Since our API design is relatively simple, there is no need to bloat out our project with an additional dependency to solve routing for us. Instead, we'll roll our own very simple adhoc routing using normal Go code. We can use a simple switch statement to detect which HTTP method was used and our `pathParams` helper function to see whether an ID was specified before passing execution to the appropriate place.

Create a new file called `handle_questions.go` and add the following `http.HandlerFunc` function:

```
func handleQuestions(w http.ResponseWriter, r *http.Request) {
    switch r.Method {
    case "POST":
        handleQuestionCreate(w, r)
    case "GET":
        params := pathParams(r, "/api/questions/:id")
        questionID, ok := params[":id"]
        if ok { // GET /api/questions/ID
            handleQuestionGet(w, r, questionID)
            return
        }
        handleTopQuestions(w, r) // GET /api/questions/
    default:
        http.NotFound(w, r)
    }
}
```

If the HTTP method is `POST`, then we'll call `handleQuestionCreate`. If it's `GET`, then we'll see whether we can extract the ID from the path and call `handleQuestionGet` if we can, or `handleTopQuestions` if we cannot.

Context in Google App Engine

If you remember, all of our calls to App Engine functions took a `context.Context` object as the first parameter, but what is that and how do we create one?

`Context` is actually an interface that provides cancelation signals, execution deadlines, and request-scoped data throughout a stack of function calls across many components and API boundaries. The Google App Engine SDK for Go uses it throughout its APIs, the details of which are kept internal to the package, which means that we (as users of the SDK) don't have to worry about it. This is a good goal for when you use `Context` in your own packages; ideally, the complexity should be kept internal and hidden.



You can, and should, learn more about Context through various online resources, starting with the *Go Concurrency Patterns: Context* blog post at <https://blog.golang.org/context>.

To create a context suitable for App Engine calls, you use the `appengine.NewContext` function, which takes `http.Request` as an argument to which the context will belong.

Underneath the routing code we just added, let's add the handler that will be responsible for creating a question, and we can see how we will create a new context for each request:

```
func handleQuestionCreate(w http.ResponseWriter, r *http.Request) {
    ctx := appengine.NewContext(r)
    var q Question
    err := decode(r, &q)
    if err != nil {
        respondErr(ctx, w, r, err, http.StatusBadRequest)
        return
    }
    err = q.Create(ctx)
    if err != nil {
        respondErr(ctx, w, r, err, http.StatusInternalServerError)
        return
    }
    respond(ctx, w, r, q, http.StatusCreated)
}
```

We create `Context` and store it in the `ctx` variable, which has become somewhat an accepted pattern throughout the Go community. We then decode our `Question` (which, due to the `OK` method, will also validate it for us) before calling the `Create` helper method that we wrote earlier. Every step of the way, we pass our context along.

If anything goes wrong, we make a call out to our `respondErr` function, which will write out the response to the client before returning and exiting early from the function.

If all is well, we respond with `Question` and a `http.StatusCreated` status code (201).

Decoding key strings

Since we are exposing the `datastore.Key` objects as the `id` field in our objects (via the `json` field tags), we expect users of our API to pass back these same ID strings when referring to specific objects. This means that we need to decode these strings and turn them back into `datastore.Key` objects. Luckily, the `datastore` package provides the answer in the form of the `datastore.DecodeKey` function.

At the bottom of `handle_questions.go`, add the following handle function to get a single question:

```
func handleQuestionGet(w http.ResponseWriter, r *http.Request,
    questionID string) {
    ctx := appengine.NewContext(r)
    questionKey, err := datastore.DecodeKey(questionID)
    if err != nil {
        respondErr(ctx, w, r, err, http.StatusBadRequest)
        return
    }
    question, err := GetQuestion(ctx, questionKey)
    if err != nil {
        if err == datastore.ErrNoSuchEntity {
            respondErr(ctx, w, r, datastore.ErrNoSuchEntity,
                http.StatusNotFound)
            return
        }
        respondErr(ctx, w, r, err, http.StatusInternalServerError)
        return
    }
    respond(ctx, w, r, question, http.StatusOK)
}
```

After we create Context again, we decode the `question ID` argument to turn the string back into a `datastore.Key` object. The `question ID` string is passed in from our routing handler code, which we added at the top of the file.

Assuming `question ID` is a valid key and the SDK was successfully able to turn it into `datastore.Key`, we call our `GetQuestion` helper function to load `Question`. If we get the `datastore.ErrNoSuchEntity` error, then we respond with a 404 (not found) status; otherwise, we'll report the error with a `http.StatusInternalServerError` code.



When writing APIs, check out the HTTP status codes and other HTTP standards and see whether you can make use of them. Developers are used to them and your API will feel more natural if it speaks the same language.

If we are able to load the question, we call `respond` and send it back to the client as JSON.

Next, we are going to expose the functionality related to answers via a similar API to the one we used for questions:

HTTP request	Description
POST /answers	Submit an answer
GET /answers	Get the answers with the specified question ID

Create a new file called `handle_answers.go` and add the routing `http.HandlerFunc` function:

```
func handleAnswers(w http.ResponseWriter, r *http.Request) {
    switch r.Method {
    case "GET":
        handleAnswersGet(w, r)
    case "POST":
        handleAnswerCreate(w, r)
    default:
        http.NotFound(w, r)
    }
}
```

For GET requests, we call `handleAnswersGet`; for POST requests, we call `handleAnswerCreate`. By default, we'll respond with a 404 Not Found response.

Using query parameters

As an alternative to parsing the path, you can just take query parameters from the URL in the request, which we will do when we add the handler that reads answers:

```
func handleAnswersGet(w http.ResponseWriter, r *http.Request) {
    ctx := appengine.NewContext(r)
    q := r.URL.Query()
    questionIDStr := q.Get("question_id")
    questionKey, err := datastore.DecodeKey(questionIDStr)
    if err != nil {
        respondErr(ctx, w, r, err, http.StatusBadRequest)
        return
    }
    answers, err := GetAnswers(ctx, questionKey)
    if err != nil {
        respondErr(ctx, w, r, err, http.StatusInternalServerError)
        return
    }
    respond(ctx, w, r, answers, http.StatusOK)
}
```

Here, we use `r.URL.Query()` to get the `http.Values` that contains the query parameters and use the `Get` method to pull out `question_id`. So, the API call will look like this:

`/api/answers?question_id=abc123`



You should be consistent in your API in the real world. We have used a mix of path parameters and query parameters to show off the differences, but it is recommended that you pick one style and stick to it.

Anonymous structs for request data

The API for answering a question is to post to `/api/answers` with a body that contains the answer details as well as the question ID string. This structure is not the same as our internal representation of `Answer` because the question ID string would need to be decoded into `datastore.Key`. We could leave the field in and indicate with field tags that it should be omitted from both the JSON and the data store, but there is a cleaner approach.

We can specify an inline, anonymous structure to hold the new answer, and the best place to do this is inside the handler function that deals with that data this means that we don't need to add a new type to our API, but we can still represent the request data we are expecting.

At the bottom of `handle_answers.go`, add the `handleAnswerCreate` function:

```
func handleAnswerCreate(w http.ResponseWriter, r *http.Request) {
    ctx := appengine.NewContext(r)
    var newAnswer struct {
        Answer
        QuestionID string `json:"question_id"`
    }
    err := decode(r, &newAnswer)
    if err != nil {
        respondErr(ctx, w, r, err, http.StatusBadRequest)
        return
    }
    questionKey, err := datastore.DecodeKey(newAnswer.QuestionID)
    if err != nil {
        respondErr(ctx, w, r, err, http.StatusBadRequest)
        return
    }
    err = newAnswer.OK()
    if err != nil {
        respondErr(ctx, w, r, err, http.StatusBadRequest)
        return
    }
}
```

```
    }
    answer := newAnswer.Answer
    user, err := UserFromAEUser(ctx)
    if err != nil {
        respondErr(ctx, w, r, err, http.StatusBadRequest)
        return
    }
    answer.User = user.Card()
    err = answer.Create(ctx, questionKey)
    if err != nil {
        respondErr(ctx, w, r, err, http.StatusInternalServerError)
        return
    }
    respond(ctx, w, r, answer, http.StatusCreated)
}
```

Look at the somewhat unusual `var newAnswer struct` line. We are declaring a new variable called `newAnswer`, which has a type of an anonymous struct (it has no name) that contains `QuestionID` string and embeds `Answer`. We can decode the request body into this type, and we will capture any specific `Answer` fields as well as `QuestionID`. We then decode the question ID into `datastore.Key` as we did earlier, validate the answer, and set the `User` (`UserCard`) field by getting the currently authenticated user and calling the `Card` helper method.

If all is well, we call `Create`, which will do the work to save the answer to the question.

Finally, we need to expose the voting functionality in our API.

Writing self-similar code

Our voting API has only a single endpoint, a post to `/votes`. So, of course, there is no need to do any routing on this method (we could just check the method in the handler itself), but there is something to be said for writing code that is familiar and similar to other code in the same package. In our case, omitting a router might jar a little if somebody else is looking at our code and expects one after seeing the routers for questions and answers.

So let's add a simple router handler to a new file called `handle_votes.go`:

```
func handleVotes(w http.ResponseWriter, r *http.Request) {
    if r.Method != "POST" {
        http.NotFound(w, r)
        return
    }
    handleVote(w, r)
}
```

Our router just checks the method and exits early if it's not POST, before calling the handleVote function, which we will add next.

Validation methods that return an error

The OK method that we added to some of our objects is a nice way to add validation methods to our code.

We want to ensure that the incoming score value is valid (in our case, either -1 or 1), so we could write a function like this:

```
func validScore(score int) bool {
    return score == -1 || score == 1
}
```

If we used this function in a few places, we would have to keep repeating the code that explained that the score was not valid. If, however, the function returns an error, you can encapsulate that in one place.

To votes.go, add the following validScore function:

```
func validScore(score int) error {
    if score != -1 && score != 1 {
        return errors.New("invalid score")
    }
    return nil
}
```

In this version, we return `nil` if the score is valid; otherwise, we return an error that explains what is wrong.

We will make use of this validation function when we add our handleVote function to handle_votes.go:

```
func handleVote(w http.ResponseWriter, r *http.Request) {
    ctx := appengine.NewContext(r)
    var newVote struct {
        AnswerID string `json:"answer_id"`
        Score     int    `json:"score"`
    }
    err := decode(r, &newVote)
    if err != nil {
        respondErr(ctx, w, r, err, http.StatusBadRequest)
        return
    }
```

```
    err = validScore(newVote.Score)
    if err != nil {
        respondErr(ctx, w, r, err, http.StatusBadRequest)
        return
    }
    answerKey, err := datastore.DecodeKey(newVote.AnswerID)
    if err != nil {
        respondErr(ctx, w, r, errors.New("invalid answer_id"),
        http.StatusBadRequest)
        return
    }
    vote, err := CastVote(ctx, answerKey, newVote.Score)
    if err != nil {
        respondErr(ctx, w, r, err, http.StatusInternalServerError)
        return
    }
    respond(ctx, w, r, vote, http.StatusCreated)
}
```

This will look pretty familiar by now, which highlights why we put all the data access logic in a different place to our handlers; the handlers can then focus on HTTP tasks, such as decoding the request and writing the response, and leave the application specifics to the other objects.

We have also broken down the logic into distinct files, with a pattern of prefixing HTTP handler code with `handle_`, so we quickly know where to look when we want to work on a specific piece of the project.

Mapping the router handlers

Let's update our `main.go` file by changing the `init` function to map the real handlers to HTTP paths:

```
func init() {
    http.HandleFunc("/api/questions/", handleQuestions)
    http.HandleFunc("/api/answers/", handleAnswers)
    http.HandleFunc("/api/votes/", handleVotes)
}
```

You can also remove the now redundant `handleHello` handler function.

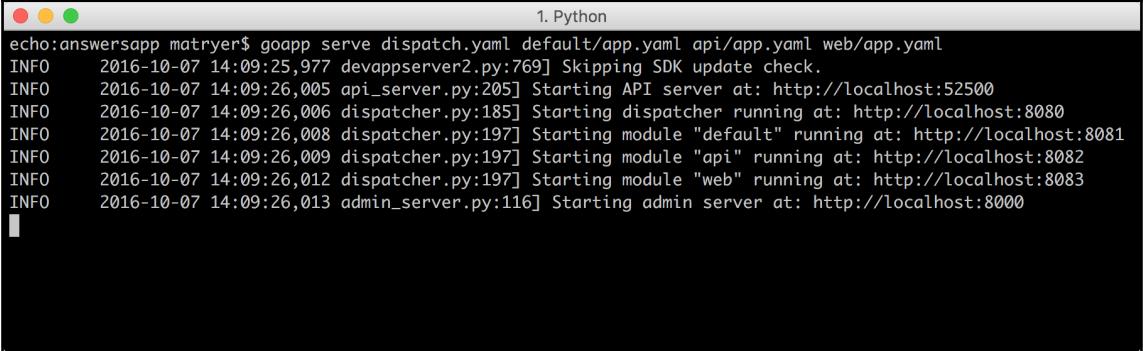
Running apps with multiple modules

For applications such as ours that have multiple modules, we need to list out all the YAML files for the `goapp` command.

To serve our new application, in a terminal, execute this:

```
goapp serve dispatch.yaml default/app.yaml api/app.yaml  
web/app.yaml
```

Starting with the dispatch file, we are listing all the associated configuration files. If you miss any, you will see an error when you try to serve your application. Here, you will notice that the output now lists that each module is being deployed on a different port:



A screenshot of a terminal window titled "1. Python". The window shows the command "echo:answersapp matryer\$ goapp serve dispatch.yaml default/app.yaml api/app.yaml web/app.yaml" followed by several "INFO" log messages. Each message indicates a server or dispatcher starting at a specific port: 52500, 8080, 8081, 8082, 8083, and 8000. The terminal has a dark background with light-colored text and standard OS X window controls.

```
echo:answersapp matryer$ goapp serve dispatch.yaml default/app.yaml api/app.yaml web/app.yaml  
INFO 2016-10-07 14:09:25,977 devappserver2.py:769] Skipping SDK update check.  
INFO 2016-10-07 14:09:26,005 api_server.py:205] Starting API server at: http://localhost:52500  
INFO 2016-10-07 14:09:26,006 dispatcher.py:185] Starting dispatcher running at: http://localhost:8080  
INFO 2016-10-07 14:09:26,008 dispatcher.py:197] Starting module "default" running at: http://localhost:8081  
INFO 2016-10-07 14:09:26,009 dispatcher.py:197] Starting module "api" running at: http://localhost:8082  
INFO 2016-10-07 14:09:26,012 dispatcher.py:197] Starting module "web" running at: http://localhost:8083  
INFO 2016-10-07 14:09:26,013 admin_server.py:116] Starting admin server at: http://localhost:8000
```

We can access modules directly by visiting each port, but luckily we have our dispatcher running on port :8080, which will do that for us based on the rules we specified in our `dispatch.yaml` configuration file.

Testing locally

Now that we have built our application, head over to `localhost:8080` to see it in action. Use the features of the application by performing the following steps:

1. Log in using your real e-mail address (that way, you'll see your Gravatar picture).
2. Ask a question.
3. Submit a couple of answers.
4. Vote the answers up and down and see the scores changing.
5. Open another browser and sign in as someone else to see what the application looks like from their point of view.

Using the admin console

The admin console is running alongside our application and is accessible at `localhost:8000`:

The screenshot shows a web browser window for the Google App Engine Development SDK 1.9.35. The URL is `localhost:8000/datastore?kind=Question`. The left sidebar lists various tools: Instances, Datastore Viewer (which is selected and highlighted in red), Datastore Indexes, Datastore Stats, Interactive Console, Memcache Viewer, Blobstore Viewer, Task Queues, Cron Jobs, XMPP, Inbound Mail, and Full Text Search. The main content area is titled "Datastore Viewer" and shows a table of "Entity Kind" `Question`. There are two rows of data:

	Key	Write Ops	ID	Key Name	AnswersCount	CTime
<input type="checkbox"/>	ahFkZXZ...	15	5629499534213120		1	2016-08-27 14:29:02
<input type="checkbox"/>	ahFkZXZ...	2	5838406743490560		1	2016-08-28 16:11:52

Below the table are buttons for "Delete" and "Flush Memcache". A status message indicates "Results 1 - 2 of 2".

Datastore Viewer lets you inspect the data of your application. You can use it to see (and even make changes to) the questions, answers, and votes data that are being generated as you use the application.

Automatically generated indexes

You can also see which indexes have been automatically created by the development server in order to satisfy the queries your application makes. In fact, if you look in the default folder, you will notice that a new file called `index.yaml` has magically appeared. This file describes those same indexes that your application will need, and when you deploy your application, this file goes up to the cloud with it to tell Google Cloud Datastore to maintain these same indexes.

Deploying apps with multiple modules

Deploying the application is slightly more complicated with multiple modules, as the dispatcher and index files each require a dedicated deployment command.

Deploy the modules with the following:

```
goapp deploy default/app.yaml api/app.yaml web/app.yaml
```

Once the operation has finished, we can update the dispatcher using the `appcfg.py` command (which you must ensure is in your path you'll find it in the Google App Engine SDK for the Go folder we downloaded at the start of the chapter):

```
appcfg.py update_dispatch .
```

Once the dispatch has been updated, we can push the indexes to the cloud:

```
appcfg.py update_indexes -A YOUR_APPLICATION_ID_HERE ./default
```

Now that the application is deployed, we can see it in the wild by navigating to our `appspot` URL; https://YOUR_APPLICATION_ID_HERE.appspot.com/.



You might get an error that says The index for this query is not ready to serve. This is because it takes Google Cloud Datastore a little time to prepare things on the server; usually, it doesn't take more than a few minutes, so go and have a cup of coffee and try again later.

An interesting aside is that if you hit the URL with HTTPS, Google's servers will serve it using HTTP/2.

Once your application is functional, ask an interesting question and send the link to your friends to solicit answers.

Summary

In this chapter, we built a fully functional question and answer application for Google App Engine.

We learned how to use the Google App Engine SDK for Go to build and test our application locally before deploying it to the cloud, ready for our friends and family to use. The application is ready to scale if it suddenly starts getting a lot of traffic, and we can rely on the healthy quota to satisfy early traffic.

We explored how to model data in Go code, keep track of keys, and persist and query data in Google Cloud Datastore. We also explored strategies to denormalize such data in order to make it quicker to read back at scale. We saw how transactions can guarantee data integrity by ensuring that only one operation occurs at a particular point in time, allowing us to build reliable counters for the score of our answers. We used predictable data store keys to ensure that our users can only have one vote per answer, and we used incomplete keys when we wanted the data store to generate the keys for us.

A lot of the techniques explored in this chapter would apply to any kind of application that persists data and interacts over a RESTful JSON API so the skills are highly transferrable.

In the next chapter, we are going to explore modern software architecture by building a real micro-service using the Go Kit framework. There are a lot of benefits to building solutions using micro-services, and so they have become a very popular choice for large, distributed systems. Lots of companies are already running such architectures (mostly written in Go) in production, and we will look at how they do it.

10

Micro-services in Go with the Go kit Framework

Micro-services are discrete components working together to provide functionality and business logic for a larger application, usually communicating over a network protocol (such as HTTP/2 or some other binary transport) and distributed across many physical machines. Each component is isolated from the others, and they take in well-defined inputs and yield well-defined outputs. Multiple instances of the same service can run across many servers and traffic can be load balanced between them. If designed correctly, it is possible for an individual instance to fail without bringing down the whole system and for new instances to be spun up during runtime to help handle load spikes.

Go kit (refer to <https://gokit.io>) is a distributed programming toolkit for the building of applications with a micro-service architecture founded by Peter Bourgon (@peterbourgon on Twitter) and now maintained by a slice of Gophers in the open. It aims to solve many of the foundational (and sometimes boring) aspects of building such systems as well as encouraging good design patterns, allowing you to focus on the business logic that makes up your product or service.

Go kit doesn't try to solve every problem from scratch; rather, it integrates with many popular related services to solve **SOA (service-oriented architecture)** problems, such as service discovery, metrics, monitoring, logging, load balancing, circuit breaking, and many other important aspects of correctly running micro-services at scale. As we build our service by hand using Go kit, you will notice that we will write a lot of boilerplate or scaffold code in order to get things working.

For smaller products and services with a small team of developers, you may well decide it is easier to just expose a simple JSON endpoint, but Go kit really shines for larger teams, building substantial systems with many different services, each being run tens or hundreds of times within the architecture. Having consistent logging, instrumentation, distributed tracing, and each item being similar to the next means running and maintaining such a system becomes significantly easier.

"Go kit is ultimately about encouraging good design practice within a service: SOLID design, or domain-driven-design, or the hexagonal architecture, etc. It's not dogmatically any of those, but tries to make good design/software engineering tractable." —Peter Bourgon

In this chapter, we are going to build some micro-services that address various security challenges (in a project called `vault`) —upon which we would be able to build further functionality. The business logic will be kept very simple, allowing us to focus on learning the principles around building micro-service systems.



There are some alternatives to Go kit as a technology choice; most of them have a similar approach but with different priorities, syntax, and patterns. Ensure that you look around at other options before embarking on a project, but the principles you learn in this chapter will apply across the board.

Specifically, in this chapter, you will learn:

- How to hand code a micro-service using Go kit
- What gRPC is and how to use it to build servers and clients
- How to use Google's protocol buffers and associated tools to describe services and communicate in a highly efficient binary format
- How endpoints in Go kit allow us to write a single service implementation and have it exposed via multiple transport protocols
- How Go kits-included subpackages help us solve lots of common problems
- How Middleware lets us wrap endpoints to adapt their behavior without touching the implementation itself
- How to describe method calls as requests and response messages
- How to rate limit our services to protect from surges in traffic
- A few other idiomatic Go tips and tricks

Some lines of code in this chapter stretch over many lines; they are written with the overflowing content right-aligned on the next line, as shown in this example:

```
func veryLongFunctionWithLotsOfArguments(one string, two int, three
    http.Handler, four string) (bool, error) {
    log.Println("first line of the function")
}
```

The first three lines in the preceding snippet should be written as one line. Don't worry; the Go compiler will be kind enough to point out if you get this wrong.

Introducing gRPC

There are many options when it comes to how our services will communicate with each other and how clients will communicate with the services, and Go kit doesn't care (rather, it doesn't mind—it cares enough to provide implementations of many popular mechanisms). In fact, we are able to add multiple options for our users and let them decide which one they want to use. We will add support the familiar JSON over HTTP, but we are also going to introduce a new technology choice for APIs.

gRPC, short for Google's **Remote Procedure Call**, is an open source mechanism used to call code that is running remotely over a network. It uses HTTP/2 for transport and protocol buffers to represent the data that makes up services and messages.

An RPC service differs from RESTful web services because rather than making changes to data using well-defined HTTP standards, as you do with REST (POST to create something, PUT to update something, DELETE to delete something, and so on), you are triggering a remote function or method instead, passing in expected arguments and getting back one or more pieces of data in response.

To highlight the difference, imagine that we are creating a new user. In a RESTful world, we could make a request like this:

```
POST /users
{
    "name": "Mat",
    "twitter": "@matryer"
}
```

And we might get a response like this:

```
201 Created
{
  "id": 1,
  "name": "Mat",
  "twitter": "@matryer"
}
```

RESTful calls represent queries or changes to the state of resources. In an RPC world, we would use generated code instead in order to make binary serialized procedure calls that feel much more like normal methods or functions in Go.

The only other key difference between a RESTful service and a gRPC service is that rather than JSON or XML, gRPC speaks a special format called **protocol buffers**.

Protocol buffers

Protocol buffers (called `protobuf` in code) are a binary serialization format that is very small and extremely quick to encode and decode. You describe data structures in an abstract way using a declarative mini language, and generate source code (in a variety of languages) to make reading and writing the data easy for users.

You can think of protocol buffers as a modern alternative to XML, except that the definition of the data structure is separated from the content, and the content is in a binary format rather than text.

It's clear to see the benefits when you look at a real example. If we wanted to represent a person with a name in XML, we could write this:

```
<person>
  <name>MAT</name>
</person>
```

This takes up about 30 bytes (discounting whitespace). Let's see how it would look in JSON:

```
{ "name": "MAT" }
```

Now we're down to 14 bytes, but the structure is still embedded in the content (the name field is spelled out along with the value).

The equivalent content in protocol buffers would only take five bytes. The following table shows each byte, along with the first five bytes of the XML and JSON representations for comparison. The **Description** row explains the meaning of the bytes in the **Content** row, which shows the protocol buffer bytes:

Byte	1	2	3	4	5
Content	0a	03	4d	61	72
Description	Type (string)	Length (3)	M	A	T
XML	<	p	e	r	s
JSON	{	"	n	a	m

The structure definition lives in a special `.proto` file, separate from the data.

There are still plenty of cases where XML or JSON would be a better choice than protocol buffers, and file size isn't the only measure when deciding a data format to use, but for fixed schema structures and remote procedure calls or for applications running at a truly massive scale, it's a popular choice for good reasons.

Installing protocol buffers

There are some tools to compile and generate source code for protocol buffers, which you can grab from the GitHub home page of the project at <https://github.com/google/protobuf/releases>. Once you've downloaded the file, unpack it and place the `protoc` file from the bin folder into an appropriate folder on your machine: one that is mentioned in your `$PATH` environment variable.

Once the `protoc` command is ready, we'll need to add a plugin that will allow us to work with Go code. In a terminal, execute this:

```
go get -u github.com/golang/protobuf/{proto,protoc-gen-go}
```

This will install two packages that we'll make use of later.

Protocol buffers language

To define our data structure, we are going to use the third version of the protocol buffers language, known as `proto3`.

Create a new folder in your \$GOPATH called `vault`, and a subfolder called `pb` inside that. The `pb` package is where our protocol buffer definition and the generated source code will live.

We are going to define a service called `Vault`, which has two methods, `Hash` and `Validate`:

Method	Description
Hash	Generate a secure hash for a given password. The hash can be stored instead of storing the password in plain text.
Validate	Given a password and a previously generated hash, the Validate method will check to ensure that the password is correct.

Each service call has a request and response pair, which we will define as well. Inside `pb`, insert the following code into a new file called `vault.proto`:

```
syntax = "proto3";
package pb;
service Vault {
    rpc Hash(HashRequest) returns (HashResponse) {}
    rpc Validate(ValidateRequest) returns (ValidateResponse) {}
}
message HashRequest {
    string password = 1;
}
message HashResponse {
    string hash = 1;
    string err = 2;
}
message ValidateRequest {
    string password = 1;
    string hash = 2;
}
message ValidateResponse {
    bool valid = 1;
}
```

Vertical whitespace has been removed to save paper, but you are free to add spaces between each block if you think it improves readability.



The first things we specify in our file are that we are using the `proto3` syntax and the name of the package for the generated source code is `pb`.

The service block defines `Vault` and the two methods-with `HashRequest`, `HashResponse`, `ValidateRequest`, and `ValidateResponse` messages defined underneath. The lines beginning with `rpc` inside the service block indicate that our service consists of two remote procedure calls: `Hash` and `Validate`.

The fields inside a message take the following format:

```
type name = position;
```

The `type` is a string that describes the scalar value type, such as `string`, `bool`, `double`, `float`, `int32`, `int64`, and so on. The `name` is a human-readable string that describes the field, such as `hash` and `password`. The `position` is an integer that indicates where in the data stream that field appears. This is important because the content is a stream of bytes, and lining up the content to the definition is vital to being able to use the format.

Additionally, if we were to add (or even rename) fields later (one of the key design features of protocol buffers), we could do so without breaking components that expect certain fields in a specific order; they would continue to work untouched, ignoring new data and just transparently passing it along.



For a complete list of the supported types as well as a deep dive into the entire language, check out the documentation at <https://developers.google.com/protocol-buffers/docs/proto3>.

Note that each method call has an associated request and response pair. These are the messages that will be sent over the network when the remote method is called.

Since the `Hash` method takes a single `password` string argument, the `HashRequest` object contains a single `password` string field. Like normal Go functions, the responses may contain an error, which is why both `HashResponse` and `ValidateResponse` have two fields. There is no dedicated `error` interface in `proto3` like there is in Go, so we are going to turn the error into a string instead.

Generating Go code

Go doesn't understand `proto3` code, but luckily the protocol buffer compiler and Go plugin we installed earlier can translate it into something Go does understand: Go code.

In a terminal, navigate to the `pb` folder and run the following:

```
protoc vault.proto --go_out=plugins=grpc:.
```

This will generate a new file called `vault.pb.go`. Open the file and inspect its contents. It has done a lot of work for us, including defining the messages and even creating `VaultClient` and `VaultServer` types for us to use, which will allow us to consume and expose the service, respectively.



You are free to decode the rest of the generated code (the file descriptor looks especially interesting) if you are interested in the details. For now, we're going to trust that it works and use the `pb` package to build our service implementation.

Building the service

At the end of the day, whatever other dark magic is going on in our architecture, it will come down to some Go method being called, doing some work, and returning a result. So the next thing we are going to do is define and implement the Vault service itself.

Inside the `vault` folder, add the following code to a new `service.go` file:

```
// Service provides password hashing capabilities.
type Service interface {
    Hash(ctx context.Context, password string) (string,
        error)
    Validate(ctx context.Context, password, hash string)
        (bool, error)
}
```

This interface defines the service.



You might think that `VaultService` would be a better name than just `Service`, but remember that since this is a Go package, it will be seen externally as `vault.Service`, which reads nicely.

We define our two methods: `Hash` and `Validate`. Each takes `context.Context` as the first argument, followed by normal `string` arguments. The responses are normal Go types as well: `string`, `bool`, and `error`.



Some libraries may still require the old context dependency, `golang.org/x/net/context`, rather than the `context` package that was made available first in Go 1.7. Watch out for errors complaining about mixed use and make sure you're importing the right one.

Part of designing micro-services is being careful about where state is stored. Even though you will implement the methods of a service in a single file, with access to global variables, you should never use them to store the per-request or even per-service state. It's important to remember that each service is likely to be running on many physical machines multiple times, each with no access to the others' global variables.

In this spirit, we are going to implement our service using an empty struct, essentially a neat idiomatic Go trick to group methods together in order to implement an interface without storing any state in the object itself. To `service.go`, add the following struct:

```
type vaultService struct{}
```



If the implementation did require any dependencies (such as a database connection or a configuration object), you could store them inside the struct and use the method receivers in your function bodies.

Starting with tests

Where possible, starting by writing test code has many advantages that usually end up increasing the quality and maintainability of your code. We are going to write a unit test that will use our new service to hash and then validate a password.

Create a new file called `service_test.go` and add the following code:

```
package vault
import (
    "testing"
    "golang.org/x/net/context"
)
func TestHasherService(t *testing.T) {
    srv := NewService()
    ctx := context.Background()
    h, err := srv.Hash(ctx, "password")
    if err != nil {
        t.Errorf("Hash: %s", err)
    }
    ok, err := srv.Validate(ctx, "password", h)
    if err != nil {
```

```
        t.Errorf("Valid: %s", err)
    }
    if !ok {
        t.Error("expected true from Valid")
    }
    ok, err = srv.Validate(ctx, "wrong password", h)
    if err != nil {
        t.Errorf("Valid: %s", err)
    }
    if ok {
        t.Error("expected false from Valid")
    }
}
```

We will create a new service via the `NewService` method and then use it to call the `Hash` and `Validate` methods. We even test an unhappy case, where we get the password wrong and ensure that `Validate` returns `false` –otherwise, it wouldn't be very secure at all.

Constructors in Go

A **constructor** in other object-oriented languages is a special kind of function that creates instances of classes. It performs any initialization and takes in required arguments such as dependencies, among others. It is usually the only way to create an object in these languages, but it often has weird syntax or relies on naming conventions (such as the function name being the same as the class, for example).

Go doesn't have constructors; it's much simpler and just has functions, and since functions can return arguments, a constructor would just be a global function that returns a usable instance of a struct. The Go philosophy of simplicity drives these kinds of decisions for the language designers; rather than forcing people to have to learn about a new concept of constructing objects, developers only have to learn how functions work and they can build constructors with them.

Even if we aren't doing any special work in the construction of an object (such as initializing fields, validating dependencies, and so on), it is sometimes worth adding a construction function anyway. In our case, we do not want to bloat the API by exposing the `vaultService` type since we already have our `Service` interface type exposed and are hiding it inside a constructor is a nice way to achieve this.

Underneath the `vaultService` struct definition, add the `NewService` function:

```
// NewService makes a new Service.
func NewService() Service {
    return vaultService{}
```

}

Not only does this prevent us from needing to expose our internals, but if in the future we do need to do more work to prepare the `vaultService` for use, we can also do it without changing the API and, therefore, without requiring the users of our package to change anything on their end, which is a big win for API design.

Hashing and validating passwords with bcrypt

The first method we will implement in our service is `Hash`. It will take a password and generate a hash. The resulting hash can then be passed (along with a password) to the `Validate` method later, which will either confirm or deny that the password is correct.



To learn more about the correct way to store passwords in applications, check out the Coda Hale blog post on the subject at <https://codahale.com/how-to-safely-store-a-password/>.

The point of our service is to ensure that passwords never need to be stored in a database, since that's a security risk if anyone is ever able to get unauthorized access to the database. Instead, you can generate a one-way hash (it cannot be decoded) that can safely be stored, and when users attempt to authenticate, you can perform a check to see whether the password generates the same hash or not. If the hashes match, the passwords are the same; otherwise, they are not.

The `bcrypt` package provides methods that do this work for us in a secure and trustworthy way.

To `service.go`, add the `Hash` method:

```
func (vaultService) Hash(ctx context.Context, password
    string) (string, error) {
    hash, err :=
        bcrypt.GenerateFromPassword([]byte(password),
            bcrypt.DefaultCost)
    if err != nil {
        return "", err
    }
    return string(hash), nil
}
```

Ensure that you import the appropriate `bcrypt` package (try golang.org/x/crypto/bcrypt). We are essentially wrapping the `GenerateFromPassword` function to generate the hash, which we then return provided no errors occurred.

Note that the receiver in the `Hash` method is just `(vaultService)`; we don't capture the variable because there is no way we can store state on an empty struct.

Next up, let's add the `Validate` method:

```
func (vaultService) Validate(ctx context.Context,
    password, hash string) (bool, error) {
    err := bcrypt.CompareHashAndPassword([]byte(hash),
        []byte(password))
    if err != nil {
        return false, nil
    }
    return true, nil
}
```

Similar to `Hash`, we are calling `bcrypt.CompareHashAndPassword` to determine (in a secure way) whether the password is correct or not. If an error is returned, it means that something is amiss and we return `false` indicating that. Otherwise, we return `true` when the password is valid.

Modeling method calls with requests and responses

Since our service will be exposed through various transport protocols, we will need a way to model the requests and responses in and out of our service. We will do this by adding a struct for each type of message our service will accept or return.

In order for somebody to call the `Hash` method and then receive the hashed password as a response, we'll need to add the following two structures to `service.go`:

```
type hashRequest struct {
    Password string `json:"password"`
}
type hashResponse struct {
    Hash string `json:"hash"`
    Err  string `json:"err,omitempty"`
}
```

The `hashRequest` type contains a single field, the `password`, and the `hashResponse` has the resulting hash and an `Err` string field in case something goes wrong.



To model remote method calls, you essentially create a struct for the incoming arguments and a struct for the return arguments.

Before continuing, see whether you can model the same request/response pair for the `Validate` method. Look at the signature in the `Service` interface, examine the arguments it accepts, and think about what kind of responses it will need to make.

We are going to add a helper method (of type `http.DecodeRequestFunc` from Go kit) that will be able to decode the JSON body of `http.Request` to `service.go`:

```
func decodeHashRequest(ctx context.Context, r
    *http.Request) (interface{}, error) {
    var req hashRequest
    err := json.NewDecoder(r.Body).Decode(&req)
    if err != nil {
        return nil, err
    }
    return req, nil
}
```

The signature for `decodeHashRequest` is dictated by Go kit because it will later use it to decode HTTP requests on our behalf. In this function, we just use `json.Decoder` to unmarshal the JSON into our `hashRequest` type.

Next, we will add the request and response structures as well as a decode helper function for the `Validate` method:

```
type validateRequest struct {
    Password string `json:"password"`
    Hash     string `json:"hash"`
}
type validateResponse struct {
    Valid bool   `json:"valid"`
    Err   string `json:"err,omitempty"`
}
func decodeValidateRequest(ctx context.Context,
    r *http.Request) (interface{}, error) {
    var req validateRequest
    err := json.NewDecoder(r.Body).Decode(&req)
    if err != nil {
        return nil, err
    }
    return req, nil
}
```

```
    }
    return req, nil
}
```

Here, the validateRequest struct takes both Password and Hash strings, since the signature has two input arguments and returns a response containing a bool datatype called Valid or Err.

The final thing we need to do is encode the response. In this case, we can write a single method to encode both the hashResponse and validateResponse objects.

Add the following code to service.go:

```
func encodeResponse(ctx context.Context,
    w http.ResponseWriter, response interface{})
error {
    return json.NewEncoder(w).Encode(response)
}
```

Our encodeResponse method just asks json.Encoder to do the work for us. Note again that the signature is general since the response type is interface{}; this is because it's a Go kit mechanism for decoding to http.ResponseWriter.

Endpoints in Go kit

Endpoints are a special function type in Go kit that represent a single RPC method. The definition is inside the endpoint package:

```
type Endpoint func(ctx context.Context, request
    interface{})
    (response interface{}, err error)
```

An endpoint function takes context.Context and request, and it returns response or error. The request and response types are interface{}, which tells us that it is up to the implementation code to deal with the actual types when building endpoints.

Endpoints are powerful because, like http.Handler (and http.HandlerFunc), you can wrap them with generalized middleware to solve a myriad of common issues that arise when building micro-services: logging, tracing, rate limiting, error handling, and more.

Go kit solves transporting over various protocols and uses endpoints as a general way to jump from their code to ours. For example, the gRPC server will listen on a port, and when it receives the appropriate message, it will call the corresponding `Endpoint` function. Thanks to Go kit, this will all be transparent to us, as we only need to deal in Go code with our `Service` interface.

Making endpoints for service methods

In order to turn our service methods into `endpoint.Endpoint` functions, we're going to write a function that handles the incoming `hashRequest`, calls the `Hash` service method, and depending on the response, builds and returns an appropriate `hashResponse` object.

To `service.go`, add the `MakeHashEndpoint` function:

```
func MakeHashEndpoint(srv Service) endpoint.Endpoint {
    return func(ctx context.Context, request interface{}) (interface{}, error) {
        req := request.(hashRequest)
        v, err := srv.Hash(ctx, req.Password)
        if err != nil {
            return hashResponse{v, err.Error()}, nil
        }
        return hashResponse{v, ""}, nil
    }
}
```

This function takes `Service` as an argument, which means that we can generate an endpoint from any implementation of our `Service` interface. We then use a type assertion to specify that the `request` argument should, in fact, be of type `hashRequest`. We call the `Hash` method, passing in the context and `Password`, which we get from `hashRequest`. If all is well, we build `hashResponse` with the value we got back from the `Hash` method and return it.

Let's do the same for the `Validate` method:

```
func MakeValidateEndpoint(srv Service) endpoint.Endpoint {
    return func(ctx context.Context, request interface{}) (interface{}, error) {
        req := request.(validateRequest)
        v, err := srv.Validate(ctx, req.Password, req.Hash)
        if err != nil {
            return validateResponse{false, err.Error()}, nil
        }
        return validateResponse{v, ""}, nil
    }
}
```

```
    }  
}
```

Here, we are doing the same: taking the request and using it to call the method before building a response. Note that we never return an error from the `Endpoint` function.

Different levels of error

There are two main types of errors in Go kit: transport errors (network failure, timeouts, dropped connection, and so on) and business logic errors (where the infrastructure of making the request and responding was successful, but something in the logic or data wasn't correct).

If the `Hash` method returns an error, we are not going to return it as the second argument; instead, we are going to build `hashResponse`, which contains the error string (accessible via the `Error` method). This is because the error returned from an endpoint is intended to indicate a transport error, and perhaps Go kit will be configured to retry the call a few times by some middleware. If our service methods return an error, it is considered a business logic error and will probably always return the same error for the same input, so it's not worth retrying. This is why we wrap the error into the response and return it to the client so that they can deal with it.

Wrapping endpoints into a Service implementation

Another very useful trick when dealing with endpoints in Go kit is to write an implementation of our `vault.Service` interface, which just makes the necessary calls to the underlying endpoints.

To `service.go`, add the following structure:

```
type Endpoints struct {  
    HashEndpoint     endpoint.Endpoint  
    ValidateEndpoint endpoint.Endpoint  
}
```

In order to implement the `vault.Service` interface, we are going to add the two methods to our `Endpoints` structure, which will build a request object, make the request, and parse the resulting response object into the normal arguments to be returned.

Add the following Hash method:

```
func (e Endpoints) Hash(ctx context.Context, password
    string) (string, error) {
    req := hashRequest{Password: password}
    resp, err := e.HashEndpoint(ctx, req)
    if err != nil {
        return "", err
    }
    hashResp := resp.(hashResponse)
    if hashResp.Err != "" {
        return "", errors.New(hashResp.Err)
    }
    return hashResp.Hash, nil
}
```

We are calling HashEndpoint with hashRequest, which we create using the password argument before caching the general response to hashResponse and returning the Hash value from it or an error.

We will do this for the Validate method:

```
func (e Endpoints) Validate(ctx context.Context, password,
    hash string) (bool, error) {
    req := validateRequest{Password: password, Hash: hash}
    resp, err := e.ValidateEndpoint(ctx, req)
    if err != nil {
        return false, err
    }
    validateResp := resp.(validateResponse)
    if validateResp.Err != "" {
        return false, errors.New(validateResp.Err)
    }
    return validateResp.Valid, nil
}
```

These two methods will allow us to treat the endpoints we have created as though they are normal Go methods; very useful for when we actually consume our service later in this chapter.

An HTTP server in Go kit

The true value of Go kit becomes apparent when we create an HTTP server for our endpoints to hash and validate.

Create a new file called `server_http.go` and add the following code:

```
package vault
import (
    "net/http"
    httptransport "github.com/go-kit/kit/transport/http"
    "golang.org/x/net/context"
)
func NewHTTPServer(ctx context.Context, endpoints
    Endpoints) http.Handler {
    m := http.NewServeMux()
    m.Handle("/hash", httptransport.NewServer(
        ctx,
        endpoints.HashEndpoint,
        decodeHashRequest,
        encodeResponse,
    ))
    m.Handle("/validate", httptransport.NewServer(
        ctx,
        endpoints.ValidateEndpoint,
        decodeValidateRequest,
        encodeResponse,
    ))
    return m
}
```

We are importing the `github.com/go-kit/kit/transport/http` package and (since we're also importing the `net/http` package) telling Go that we're going to explicitly refer to this package as `httptransport`.

We are using the `NewServeMux` function from the standard library to build `http.Handler` interface with simple routing and mapping the `/hash` and `/validate` paths. We take the `Endpoints` object since we want our HTTP server to serve these endpoints, including any middleware that we will add later. Calling `httptransport.NewServer` is how we get Go kit to give us an HTTP handler for each endpoint. Like most functions, we pass in `context.Context` as the first argument, which will form the base context for each request. We also pass in the endpoint as well as the decoding and encoding functions that we wrote earlier so that the server knows how to unmarshal and marshal the JSON messages.

A gRPC server in Go kit

Adding a gPRC server using Go kit is almost as easy as adding a JSON/HTTP server, like we did in the last section. In our generated code (in the pb folder), we were given the following `pb.VaultServer` type:

```
type VaultServer interface {
    Hash(context.Context, *HashRequest)
        (*HashResponse, error)
    Validate(context.Context, *ValidateRequest)
        (*ValidateResponse, error)
}
```

This type is very similar to our own `Service` interface, except that it takes in generated request and response classes rather than raw arguments.

We'll start by defining a type that will implement the preceding interface. Add the following code to a new file called `server_grpc.go`:

```
package vault
import (
    "golang.org/x/net/context"
    grpctransport "github.com/go-kit/kit/transport/grpc"
)
type grpcServer struct {
    hash      grpctransport.Handler
    validate  grpctransport.Handler
}
func (s *grpcServer) Hash(ctx context.Context,
    r *pb.HashRequest) (*pb.HashResponse, error) {
    _, resp, err := s.hash.ServeGRPC(ctx, r)
    if err != nil {
        return nil, err
    }
    return resp.(*pb.HashResponse), nil
}
func (s *grpcServer) Validate(ctx context.Context,
    r *pb.ValidateRequest) (*pb.ValidateResponse, error) {
    _, resp, err := s.validate.ServeGRPC(ctx, r)
    if err != nil {
        return nil, err
    }
    return resp.(*pb.ValidateResponse), nil
}
```

Note that you'll need to import `github.com/go-kit/kit/transport/grpc` as `grpctransport`, along with the generated `pb` package.

The `grpcServer` struct contains a field for each of the service endpoints, this time of type `grpctransport.Handler`. Then, we implement the methods of the interface, calling the `ServeGRPC` method on the appropriate handler. This method will actually serve requests by first decoding them, calling the appropriate endpoint function, getting the response, and encoding it and sending it back to the client who made the request.

Translating from protocol buffer types to our types

You'll notice that we're using the request and response objects from the `pb` package, but remember that our own endpoints use the structures we added to `service.go` earlier. We are going to need a method for each type in order to translate to and from our own types.



There's a lot of repetitive typing coming up; feel free to copy and paste this from the GitHub repository at <https://github.com/matryer/goblueprint> to save your fingers. We're hand coding this manually because it's important to understand all the pieces that make up the service.

To `server_grpc.go`, add the following function:

```
func EncodeGRPCHashRequest(ctx context.Context,
    r interface{}) (interface{}, error) {
    req := r.(hashRequest)
    return &pb.HashRequest{Password: req.Password}, nil
}
```

This function is an `EncodeRequestFunc` function defined by Go kit, and it is used to translate our own `hashRequest` type into a protocol buffer type that can be used to communicate with the client. It uses `interface{}` types because it's general, but in our case, we can be sure about the types so we cast the incoming request to `hashRequest` (our own type) and then build a new `pb.HashRequest` object using the appropriate fields.

We are going to do this for both encoding and decoding requests and responses for both `hash` and `validate` endpoints. Add the following code to `server_grpc.go`:

```
func DecodeGRPCHashRequest(ctx context.Context,
    r interface{}) (interface{}, error) {
    req := r.(*pb.HashRequest)
    return hashRequest{Password: req.Password}, nil
}
```

```
}

func EncodeGRPCHashResponse(ctx context.Context,
    r interface{}) (interface{}, error) {
    res := r.(hashResponse)
    return &pb.HashResponse{Hash: res.Hash, Err: res.Err},
        nil
}

func DecodeGRPCHashResponse(ctx context.Context,
    r interface{}) (interface{}, error) {
    res := r.(*pb.HashResponse)
    return hashResponse{Hash: res.Hash, Err: res.Err}, nil
}

func EncodeGRPCValidateRequest(ctx context.Context,
    r interface{}) (interface{}, error) {
    req := r.(validateRequest)
    return &pb.ValidateRequest{Password: req.Password,
        Hash: req.Hash}, nil
}

func DecodeGRPCValidateRequest(ctx context.Context,
    r interface{}) (interface{}, error) {
    req := r.(*pb.ValidateRequest)
    return validateRequest{Password: req.Password,
        Hash: req.Hash}, nil
}

func EncodeGRPCValidateResponse(ctx context.Context,
    r interface{}) (interface{}, error) {
    res := r.(validateResponse)
    return &pb.ValidateResponse{Valid: res.Valid}, nil
}

func DecodeGRPCValidateResponse(ctx context.Context,
    r interface{}) (interface{}, error) {
    res := r.(*pb.ValidateResponse)
    return validateResponse{Valid: res.Valid}, nil
}
```

As you can see, there is a lot of boilerplate coding to do in order to get things working.

Code generation (not covered here) would have great application here, since the code is very predictable and self-similar.



The final thing to do in order to get our gRPC server working is to provide a helper function to create an instance of our `grpcServer` structure. Underneath the `grpcServer` struct, add the following code:

```
func NewGRPCServer(ctx context.Context, endpoints  
Endpoints) pb.VaultServer {  
    return &grpcServer{  
        hash: grpctransport.NewServer(  
            ctx,  
            endpoints.HashEndpoint,  
            DecodeGRPCHashRequest,  
            EncodeGRPCHashResponse,  
        ),  
        validate: grpctransport.NewServer(  
            ctx,  
            endpoints.ValidateEndpoint,  
            DecodeGRPCValidateRequest,  
            EncodeGRPCValidateResponse,  
        ),  
    }  
}
```

Like our HTTP server, we take in a base context and the actual `Endpoints` implementation that we are exposing via the gRPC server. We create and return a new instance of our `grpcServer` type, setting the handlers for both `hash` and `validate` by calling `grpctransport.NewServer`. We use our `endpoint.Endpoint` functions for our service and tell the service which of our encoding/decoding functions to use for each case.

Creating a server command

So far, all of our service code lives inside the `vault` package. We are now going to use this package to create a new tool to expose the server functionality.

Create a new folder in `vault` called `cmd`, and inside it create another called `vaultd`. We are going to put our command code inside the `vaultd` folder because even though the code will be in the `main` package, the name of the tool will be `vaultd` by default. If we just put the command in the `cmd` folder, the tool would be built into a binary called `cmd`-which is pretty confusing.



In Go projects, if the primary use of the package is to be imported into other programs (such as Go kit), then the root level files should make up the package and will have an appropriate package name (not `main`). If the primary purpose is a command-line tool, such as the Drop command (`htt`



ps://github.com/matryer/drop), then the root files will be in the main package. The rationale for this comes down to usability; when importing a package, you want the string the user has to type to be the shortest it can be. Similarly, when using go install, you want the path to be short and sweet.

The tool we are going to build (suffixed with d, indicating that it is a daemon or a background task) will spin up both our gRPC and JSON/HTTP servers. Each will run in their own goroutine, and we will trap any termination signals or errors from the servers, which will cause the termination of our program.

In Go kit, main functions end up being quite large, which is by design; there is a single function that contains the entirety of your micro-service; from there, you can dig down into the details, but it provides an at-a-glance view of each component.

We will build up the main function piece by piece inside a new main.go file in the vaultd folder, starting with the fairly big list of imports:

```
import (
    "flag"
    "fmt"
    "log"
    "net"
    "net/http"
    "os"
    "os/signal"
    "syscall"
    "your/path/to/vault"
    "your/path/to/vault/pb"
    "golang.org/x/net/context"
    "google.golang.org/grpc"
)
```

The your/path/to prefixes should be replaced with the actual route from \$GOPATH to where your project is. Pay attention to the context import too; it's quite possible that you just need to type context rather than the import listed here depending on when Go kit transitions to Go 1.7. Finally, the grpc package from Google provides everything we need in order to expose gRPC capabilities over the network.

Now, we will put together our `main` function; remember that all the sections following this one go inside the body of the `main` function:

```
func main() {
    var (
        httpAddr = flag.String("http", ":8080",
            "http listen address")
        gRPCAddr = flag.String("grpc", ":8081",
            "gRPC listen address")
    )
    flag.Parse()
    ctx := context.Background()
    srv := vault.NewService()
    errChan := make(chan error)
```

We use flags to allow the ops team to decide which endpoints we will listen on when exposing the service on the network, but provide sensible defaults of :8080 for the JSON/HTTP server and :8081 for the gRPC server.

We then create a new context using the `context.Background()` function, which returns a non-nil, empty context that has no cancelation or deadline specified and contains no values, perfect for the base context of all of our services. Requests and middleware are free to create new context objects from this one in order to add request-scoped data or deadlines.

Next, we use our `NewService` constructor to make a new `Service` type for us and make a zero-buffer channel, which can take an error should one occur.

We will now add the code that traps termination signals (such as *Ctrl + C*) and sends an error down `errChan`:

```
go func() {
    c := make(chan os.Signal, 1)
    signal.Notify(c, syscall.SIGINT, syscall.SIGTERM)
    errChan <- fmt.Errorf("%s", <-c)
}()
```

Here, in a new goroutine, we ask `signal.Notify` to tell us when we receive the `SIGINT` or `SIGTERM` signals. When that happens, the signal will be sent down the `c` channel, at which point we'll format it as a string (its `String()` method will be called), and we turn that into an error, which we'll send down `errChan`, resulting in the termination of the program.

Using Go kit endpoints

It is time to create one of our endpoints instances that we can pass to our servers. Add the following code to the main function body:

```
hashEndpoint := vault.MakeHashEndpoint(srv)
validateEndpoint := vault.MakeValidateEndpoint(srv)
endpoints := vault.Endpoints{
    HashEndpoint: hashEndpoint,
    ValidateEndpoint: validateEndpoint,
}
```

We are assigning the fields to the output of our endpoint helper functions for both the hash and validate methods. We are passing in the same service for both, so the `endpoints` variable essentially ends up being a wrapper around our `srv` service.



You may be tempted to neaten up this code by removing the assignment to the variables altogether and just set the return of the helper functions to the fields in the struct initialization, but when we come to add middleware later, you'll be thankful for this approach.

We are now ready to start up our JSON/HTTP and gRPC servers using these endpoints.

Running the HTTP server

Now we will add the goroutine to make and run the JSON/HTTP server to the main function body:

```
// HTTP transport
go func() {
    log.Println("http:", *httpAddr)
    handler := vault.NewHTTPServer(ctx, endpoints)
    errChan <- http.ListenAndServe(*httpAddr, handler)
}()
```

All the heavy lifting has already been done for us in our package code by Go kit, so we are left with simply calling the `NewHTTPServer` function, passing in the background context and the service endpoints we wish for it to expose, before calling the standard library's `http.ListenAndServe`, which exposes the handler functionality in the specified `httpAddr`. If an error occurs, we send it down the error channel.

Running the gRPC server

There is a little more work to do in order to run the gRPC server, but it is still pretty simple. We must create a low-level TCP network listener and serve the gRPC server over that. Add the following code to the main function body:

```
go func() {
    listener, err := net.Listen("tcp", *gRPCAddr)
    if err != nil {
        errChan <- err
        return
    }
    log.Println("grpc:", *gRPCAddr)
    handler := vault.NewGRPCServer(ctx, endpoints)
    gRPCServer := grpc.NewServer()
    pb.RegisterVaultServer(gRPCServer, handler)
    errChan <- gRPCServer.Serve(listener)
}()
```

We make the TCP listener on the `gRPCAddr` endpoint specified, sending any errors down the `errChan` error channel. We use `vault.NewGRPCServer` to create the handler, again passing in the background context and the instance of `Endpoints` we are exposing.



Note how both the JSON/HTTP server and the gRPC server are actually exposing the same service—literally the same instance.

We then create a new gRPC server from Google's `grpc` package and register it using our own generated `pb` package via the `RegisterVaultServer` function.



The `RegisterVaultService` function just calls `RegisterService` on our `grpcServer` but hides the internals of the service description that was automatically generated. If you look in `vault.pb.go` and search for the `RegisterVaultServer` function, you will see that it makes a reference to something like `&_Vault_serviceDesc`, which is the description of the service. Feel free to dig around the generated code; the metadata is especially interesting, but out of scope for this book.

We then ask the server to `Serve` itself, throwing any errors down the same error channel if they occur.



It's out of scope for this chapter, but it is recommended that every service be delivered with **Transport Layer Security (TLS)**, especially the ones dealing with passwords.

Preventing a main function from terminating immediately

If we closed our main function here, it would immediately exit and terminate all of our servers. This is because everything we're doing that would prevent this is inside its own goroutine. To prevent this, we need a way to block the function at the end to wait until something tells the program to terminate.

Since we are using the `<-errChan` error channel for errors, this is a perfect candidate. We can just listen on this channel, which (while nothing has been sent down it) will block and allow the other goroutines to do their work. If something goes wrong (or if a termination signal is received), the `<-errChan` call will unblock and exit and all goroutines will be stopped.

At the bottom of the main function, add the final statement and closing block:

```
    log.Fatalln(<-errChan)  
}
```

When an error occurs, we'll just log it and exit with a nonzero code.

Consuming the service over HTTP

Now that we have wired everything up, we can test the HTTP server using the `curl` command—or any tool that lets us make JSON/HTTP requests.

In a terminal, let's start by running our servers. Head over to the `vault/cmd/vaultd` folder and start the program:

```
go run main.go
```

Once the server is running, you'll see something like this:

```
http: :8080  
grpc: :8081
```

Now, open another terminal and issue the following HTTP request using curl:

```
curl -XPOST -d '{"password":"hernandez"}'  
http://localhost:8080/hash
```

We are making a POST request to the hash endpoint with a JSON body that contains the password we want for hashing. Then, we get something like this:

```
{"hash": "$2a$10$IXYT10DuK3Hu.  
NZQsyNaff1tyxe5QkYZKM5by/5Ren"}
```



The hash in this example won't match yours—there are many acceptable hashes and there's no way to know which one you'll get. Ensure that you copy and paste your actual hash (everything inside the double quotes).

The resulting hash is what we would store in our data store given the specified password. Then, when the user tries to log in again, we will make a request with the password they entered, along with this hash, to the validate endpoint:

```
curl -XPOST -d  
'{"password":"hernandez",  
 "hash":"PASTE_YOUR_HASH_HERE"}'  
http://localhost:8080/validate
```

Make this request by copying and pasting the correct hash and entering the same hernandez password, and you will see this result:

```
{"valid":true}
```

Now, change the password (this is equivalent to the user getting it wrong) and you will see this:

```
{"valid":false}
```

You can see that the JSON/HTTP micro-service exposure for our vault service is complete and working.

Next, we will look at how we can consume the gRPC version.

Building a gRPC client

Unlike JSON/HTTP services, gRPC services aren't easy for humans to interact with. They're really intended as machine-to-machine protocols, and so we must write a program if we wish to use them.

To help us do this, we are first going to add a new package inside our vault service called `vault/client/grpc`. It will, given a gRPC client connection object that we get from Google's `grpc` package, provide an object that performs the appropriate calls, encoding and decoding, for us, all hidden behind our own `vault.Service` interface. So, we will be able to use the object as though it is just another implementation of our interface.

Create new folders inside `vault` so that you have the path of `vault/client/grpc`. You can imagine adding other clients if you so wish, so this seems a good pattern to establish.

Add the following code to a new `client.go` file:

```
func New(conn *grpc.ClientConn) vault.Service {
    var hashEndpoint = grpctransport.NewClient(
        conn, "Vault", "Hash",
        vault.EncodeGRPCHashRequest,
        vault.DecodeGRPCHashResponse,
        pb.HashResponse{},
    ).Endpoint()
    var validateEndpoint = grpctransport.NewClient(
        conn, "Vault", "Validate",
        vault.EncodeGRPCValidateRequest,
        vault.DecodeGRPCValidateResponse,
        pb.ValidateResponse{},
    ).Endpoint()
    return vault.Endpoints{
        HashEndpoint: hashEndpoint,
        ValidateEndpoint: validateEndpoint,
    }
}
```

The `grpctransport` package is referring to github.com/go-kit/kit/transport/grpc. This might feel familiar by now; we are making two new endpoints based on the specified connection, this time being explicit about the `Vault` service name and the endpoint names `Hash` and `Validate`. We pass in appropriate encoders and decoders from our `vault` package and empty response objects before wrapping them both in our `vault.Endpoints` structure that we added—the one that implements the `vault.Service` interface that just triggers the specified endpoints for us.

A command-line tool to consume the service

In this section, we are going to write a command-line tool (or CLI-command-line interface), which will allow us to communicate with our service through the gRPC protocol. If we were writing another service in Go, we would use the vault client package in the same way as we will when we write our CLI tool.

Our tool will let you access the services in a fluent way on the command line by separating commands and arguments with spaces such that we can hash a password like this:

```
vaultcli hash MyPassword
```

We will be able to validate a password with a hash like this:

```
vaultcli hash MyPassword HASH_Goes_Here
```

In the cmd folder, create a new folder called `vaultcli`. Add a `main.go` file and insert the following main function:

```
func main() {
    var (
        grpcAddr = flag.String("addr", ":8081",
            "gRPC address")
    )
    flag.Parse()
    ctx := context.Background()
    conn, err := grpc.Dial(*grpcAddr, grpc.WithInsecure(),
        grpc.WithTimeout(1*time.Second))
    if err != nil {
        log.Fatalln("gRPC dial:", err)
    }
    defer conn.Close()
    vaultService := grpcclient.New(conn)
    args := flag.Args()
    var cmd string
    cmd, args = pop(args)
    switch cmd {
    case "hash":
        var password string
        password, args = pop(args)
        hash(ctx, vaultService, password)
    case "validate":
        var password, hash string
        password, args = pop(args)
        hash, args = pop(args)
        validate(ctx, vaultService, password, hash)
    default:
    }
}
```

```
    log.Fatalln("unknown command", cmd)
}
}
```

Ensure that you import the `vault/client/grpc` package as `grpcclient` and `google.golang.org/grpc` as `grpc`. You'll also need to import the `vault` package.

We parse the flags and get a background context as usual before dialing the gRPC endpoint to establish a connection. If all is well, we defer the closing of the connection and create our vault service client using that connection. Remember that this object implements our `vault.Service` interface, so we can just call the methods as though they were normal Go methods, without worrying about the fact that communication is taking place over a network protocol.

Then, we start parsing the command-line arguments in order to decide which execution flow to take.

Parsing arguments in CLIs

Parsing arguments in command-line tools is very common, and there is a neat idiomatic way to do it in Go. The arguments are all available via the `os.Args` slice, or if you're using flags, the `flags.Args()` method (which gets arguments with flags stripped). We want to take each argument off the slice (from the beginning) and consume them in an order, which will help us decide which execution flow to take through the program. We're going to add a helper function called `pop`, which will return the first item, and the slice with the first item trimmed.

We'll write a quick unit test to ensure that our `pop` function is working as expected. If you would like to try and write the `pop` function yourself, then you should do that once the test is in place. Remember that you can run tests by navigating to the appropriate folder in a terminal and executing this:

```
go test
```

Create a new file inside `vaultcli` called `main_test.go` and add the following test function:

```
func TestPop(t *testing.T) {
    args := []string{"one", "two", "three"}
    var s string
    s, args = pop(args)
    if s != "one" {
        t.Errorf("unexpected \"%s\"", s)
```

```
        }
        s, args = pop(args)
        if s != "two" {
            t.Errorf("unexpected \"%s\"", s)
        }
        s, args = pop(args)
        if s != "three" {
            t.Errorf("unexpected \"%s\"", s)
        }
        s, args = pop(args)
        if s != "" {
            t.Errorf("unexpected \"%s\"", s)
        }
    }
```

We expect each call to `pop` to yield the next item in the slice and empty arguments once the slice is empty.

At the bottom of `main.go`, add the `pop` function:

```
func pop(s []string) (string, []string) {
    if len(s) == 0 {
        return "", s
    }
    return s[0], s[1:]
}
```

Maintaining good line of sight by extracting case bodies

The only thing that remains for us to do is implement the `hash` and `validate` methods referred to in the `switch` statement shown earlier.

We could have embedded this code inside the `switch` statement itself, but that would make the `main` function very difficult to read and also hide happy path execution at different indentation levels, something we should try to avoid.

Instead, it is a good practice to have the cases inside the `switch` statement jump out to a dedicated function, taking in any arguments it needs. Underneath the `main` function, add the following `hash` and `validate` functions:

```
func hash(ctx context.Context, service vault.Service,
    password string) {
    h, err := service.Hash(ctx, password)
    if err != nil {
```

```
        log.Fatalln(err.Error())
    }
    fmt.Println(h)
}
func validate(ctx context.Context, service vault.Service,
    password, hash string) {
    valid, err := service.Validate(ctx, password, hash)
    if err != nil {
        log.Fatalln(err.Error())
    }
    if !valid {
        fmt.Println("invalid")
        os.Exit(1)
    }
    fmt.Println("valid")
}
```

These functions simply call the appropriate method on the service, and depending on the result, log or print the results to the console. If the validate method returns false, the program will exit with an exit code of 1, since nonzero means an error.

Installing tools from the Go source code

To install the tool, we just have to navigate to the `vaultcli` folder in a terminal and type this:

```
go install
```

Provided there are no errors, the package will be built and deployed to the `$GOPATH/bin` folder, which should already be listed in your `$PATH` environment variable. This means that the tool is ready for use just like a normal command in your terminal.

The name of the binary that is deployed will match the folder name, and this is why we have an additional folder inside the `cmd` folder even if we are only building a single command.

Once you have installed the command, we can use it to test the gRPC server.

Head over to `cmd/vaultd` and start the server (if it isn't already running) by typing the following:

```
go run main.go
```

In another terminal, let's hash a password by typing this:

```
vaultcli hash blanca
```

Note that the hash is returned. Now let's validate this hash:

```
vaultcli validate blanca PASTE_HASH_HERE
```



The hash may contain special characters that interfere with your terminal, so you should escape the string with quotes if required. On a Mac, format the argument with `$ 'PASTE_HASH_HERE'` to properly escape it. On Windows, try surrounding the argument with exclamation points: `!PASTE_HASH_HERE!`.

If you get the password right, you'll notice that you see the word `valid`; otherwise, you'll see `invalid`.

Rate limiting with service middleware

Now that we have built a complete service, we are going to see how easy it is to add middleware to our endpoints in order to extend the service without touching the actual implementations themselves.

In real-world services, it is sensible to limit the number of requests it will attempt to handle so that the service doesn't get overwhelmed. This can happen if the process needs more memory than is available, or we might notice performance degradation if it eats up too much of the CPU. In a micro-service architecture, the strategy to solving these problems is to add another node and spread the load, which means that we want each individual instance to be rate limited.

Since we are providing the client, we should add rate limiting there, which would prevent too many requests from getting on the network. But it is also sensible to add rate limiting to the server in case many clients are trying to access the same services at the same time. Luckily, endpoints in Go kit are used for both the client and server, so we can use the same code to add middleware in both places.

We are going to add a **Token Bucket**-based rate limiter, which you can read more about at https://en.wikipedia.org/wiki/Token_bucket. The guys at Juju have written a Go implementation that we can use by importing `github.com/juju/ratelimit`, and Go kit has middleware built for this very implementation, which will save us a lot of time and effort.

The general idea is that we have a bucket of tokens, and each request will need a token in order to do its work. If there are no tokens in the bucket, we have reached our limit and the request cannot be completed. Buckets refill over time at a specific interval.

Import `github.com/juju/ratelimit` and before we create our `hashEndpoint`, insert the following code:

```
rlbucket := ratelimit.NewBucket(1*time.Second, 5)
```

The `NewBucket` function creates a new rate limiting bucket that will refill at a rate of one token per second, up to a maximum of five tokens. These numbers are pretty silly for our case, but we want to be able to reach our limits manually during the development.

Since the Go kit `ratelimit` package has the same name as the Juju one, we are going to need to import it with a different name:

```
import ratelimitkit "github.com/go-kit/kit/ratelimit"
```

Middleware in Go kit

Endpoint middleware in Go kit is specified by the `endpoint.Middleware` function type:

```
type Middleware func(Endpoint) Endpoint
```

A piece of middleware is simply a function that takes `Endpoint` and returns `Endpoint`. Remember that `Endpoint` is also a function:

```
type Endpoint func(ctx context.Context, request  
interface{}) (response interface{}, err error)
```

This gets a little confusing, but they are the same as the wrappers we built for `http.HandlerFunc`. A middleware function returns an `Endpoint` function that does something before and/or after calling the `Endpoint` being wrapped. The arguments passed into the function that returns the `Middleware` are closed in, which means that they are available to the inner code (via closures) without the state having to be stored anywhere else.

We are going to use the `NewTokenBucketLimiter` middleware from Go kit's `ratelimit` package, and if we take a look at the code, we'll see how it uses closures and returns functions to inject a call to the token bucket's `TakeAvailable` method before passing execution to the next endpoint:

```
func NewTokenBucketLimiter(tb *ratelimit.Bucket)
    endpoint.Middleware {
    return func(next endpoint.Endpoint) endpoint.Endpoint {
        return func(ctx context.Context, request interface{}) (interface{}, error) {
            if tb.TakeAvailable(1) == 0 {
                return nil, ErrLimited
            }
            return next(ctx, request)
        }
    }
}
```

A pattern has emerged within Go kit where you obtain the endpoint and then put all middleware adaptations inside their own block immediately afterwards. The returned function is given the endpoint when it is called, and the same variable is overwritten with the result.

For a simple example, consider this code:

```
e := getEndpoint(srv)
{
    e = getSomeMiddleware()(e)
    e = getLoggingMiddleware(logger)(e)
    e = getAnotherMiddleware(something)(e)
}
```

We will now do this for our endpoints; update the code inside the main function to add the rate limiting middleware:

```
hashEndpoint := vault.MakeHashEndpoint(srv)
{
    hashEndpoint = ratelimitkit.NewTokenBucketLimiter
        (rlbucket)(hashEndpoint)
}
validateEndpoint := vault.MakeValidateEndpoint(srv)
{
    validateEndpoint = ratelimitkit.NewTokenBucketLimiter
        (rlbucket)(validateEndpoint)
}
endpoints := vault.Endpoints{
    HashEndpoint:      hashEndpoint,
```

```
    ValidateEndpoint: validateEndpoint,  
}
```

There's nothing much to change here; we're just updating the `hashEndpoint` and `validateEndpoint` variables before assigning them to the `vault.Endpoints` struct.

Manually testing the rate limiter

To see whether our rate limiter is working, and since we set such low thresholds, we can test it just using our command-line tool.

First, restart the server (so the new code runs) by hitting *Ctrl + C* in the terminal window running the server. This signal will be trapped by our code, and an error will be sent down `errChan`, causing the program to quit. Once it has terminated, restart it:

```
go run main.go
```

Now, in another window, let's hash some passwords:

```
vaultcli hash bourgon
```

Repeat this command a few times—in most terminals, you can press the up arrow key and return. You'll notice that the first few requests succeed because it's within the limits, but if you get a little more aggressive and issue more than five requests in a second, you'll notice that we get errors:

```
$ vaultcli hash bourgon  
$2a$10$q3NTkjG0YFZhTG6gBU2WpenFmNzdN74oX0MDSTryiAqRXJ7RVw9sy  
$ vaultcli hash bourgon  
$2a$10$CdEEtxSDUyJEIFaykbMM1.EikxvV5921gs./.7If6V0dh2x0Q1oLW  
$ vaultcli hash bourgon  
$2a$10$1DSqQJJGCmVOptwIx6rrSOZwLlOhjHNC83OPVE8SdQ9q73Li5x2le  
$ vaultcli hash bourgon  
Invoke: rpc error: code = 2 desc = rate limit exceeded  
$ vaultcli hash bourgon  
Invoke: rpc error: code = 2 desc = rate limit exceeded  
$ vaultcli hash bourgon  
Invoke: rpc error: code = 2 desc = rate limit exceeded  
$ vaultcli hash bourgon  
$2a$10$kriTDXdyT6J4IrqZLwgBde663nLhoG3innhCNuf8H2nHf7kxnmSza
```

This shows that our rate limiter is working. We see errors until the token bucket fills back up, where our requests are fulfilled again.

Graceful rate limiting

Rather than returning an error (which is a pretty harsh response), perhaps we would prefer the server to just hold onto our request and fulfill it when it can-called throttling. For this case, Go kit provides the `NewTokenBucketThrottler` middleware.

Update the middleware code to use this middleware function instead:

```
hashEndpoint := vault.MakeHashEndpoint(srv)
{
    hashEndpoint = ratelimitkit.NewTokenBucketThrottler(rlbucket,
        time.Sleep)(hashEndpoint)
}
validateEndpoint := vault.MakeValidateEndpoint(srv)
{
    validateEndpoint = ratelimitkit.NewTokenBucketThrottler(rlbucket,
        time.Sleep)(validateEndpoint)
}
endpoints := vault.Endpoints{
    HashEndpoint:    hashEndpoint,
    ValidateEndpoint: validateEndpoint,
}
```

The first argument to `NewTokenBucketThrottler` is the same endpoint as earlier, but now we have added a second argument of `time.Sleep`.



Go kit allows us to customize the behavior by specifying what should happen when the delay needs to take place. In our case, we're passing `time.Sleep`, which is a function that will ask execution to pause for the specified amount of time. You could write your own function here if you wanted to do something different, but this works for now.

Now repeat the test from earlier, but this time, note that we never get an error-instead, the terminal will hang for a second until the request can be fulfilled.

Summary

We covered a lot through this chapter as we put together a real example of a micro-service. There is a lot of work involved without code generation, but the benefits for large teams and big micro-service architectures pay for the investment as you build self-similar, discrete components that make up the system.

We learned how gRPC and protocol buffers give us highly efficient transport communications between clients and servers. Using the `proto3` language, we defined our service, including messages, and used the tools to generate a Go package that provided the client and server code for us.

We explored the fundamentals of Go kit and how we can use endpoints to describe the methods of our services. We let Go kit do the heavy lifting for us when it came to building HTTP and gRPC servers by making use of the packages included in the project. We saw how middleware functions let us easily adapt our endpoints to, among other things, rate limit the amount of traffic the server will have to handle.

We also learned about constructors in Go, a neat trick to parse incoming command-line arguments, and how to hash and validate passwords using the `bcrypt` package, which is a sensible approach that helps us avoid storing passwords at all.

There is a lot more to building micro-services, and it is recommended that you head over to the Go kit website at <https://gokit.io> or join the conversation on the `#go-kit` slack channel at gophers.slack.com to learn more.

Now that we have built our Vault service, we need to think about our options in order to deploy it into the wild. In the next chapter, we'll package our micro-service into a Docker container and deploy it to Digital Ocean's cloud.

11

Deploying Go Applications Using Docker

Docker is an open source ecosystem (technology and range of associated services) that allows you to package applications into containers that are simple, lightweight, and portable; they will run in the same way regardless of which environment they run on. This is useful when you consider that our development environment (perhaps a Mac) is different from a production environment (such as a Linux server or even a cloud service) and that there is a large number of different places that we might want to deploy the same application.

Most cloud platforms already support Docker, which makes it a great option to deploy our apps into the wild.

In Chapter 9, *Building a Q&A Application for Google App Engine*, we built an application for Google App Engine. We would need to make significant changes to our code if we decided that we wanted to run our application on a different platform even if we forgot about our use of Google Cloud Datastore. Building applications with a mind to deploying them within Docker containers gives us an additional level of flexibility.



Did you know that Docker itself was written in Go? See for yourself by browsing the source code at <https://github.com/docker/docker>.

In this chapter, you will learn:

- How to write a simple Dockerfile to describe an application
- How to use the `docker` command to build the container
- How to run Docker containers locally and terminate them

- How to deploy Docker containers to Digital Ocean
- How to use the features in Digital Ocean to spin up instances that already have Docker preconfigured

We are going to put the Vault service we created in Chapter 10, *Micro-services in Go with the Go kit Framework*, into a Docker image and deploy it to the cloud.

Using Docker locally

Before we can deploy our code to the cloud, we must use the Docker tools on our development machine to build and push the image to Docker Hub.

Installing Docker tools

In order to build and run containers, you need to install Docker on your development machine. Head over to <https://www.docker.com/products/docker> and download the appropriate installer for your computer.

Docker and its ecosystem are evolving rapidly, so it is a good idea to make sure you're up to date with the latest release. Similarly, it is possible that some details will change in this chapter; if you get stuck, visit the project home page at <https://github.com/matryer/goblueprints> for some helpful tips.

Dockerfile

A Docker image is like a mini virtual machine. It contains everything that's needed to run an application: the operating system the code will run on, any dependencies that our code might have (such as Go kit in the case of our Vault service), and the binaries of our application itself.

An image is described with `Dockerfile`; a text file containing a list of special commands that instruct Docker how to build the image. They are usually based on another container, which saves you from building up everything that might be needed in order to build and run Go applications.

Inside the `vault` folder from the code we wrote in Chapter 10, *Micro-services in Go with the Go kit Framework*, add a file called `Dockerfile` (note that this filename has no extension), containing the following code:

```
FROM scratch
MAINTAINER Your Name <your@email.address>
ADD vaultd vaultd
EXPOSE 8080 8081
ENTRYPOINT ["/vaultd"]
```

Each line in a `Dockerfile` file represents a different command that is run while the image is being built. The following table describes each of the commands we have used:

Command	Description
FROM	The name of the image that this image will be based on. Single words, such as <code>scratch</code> , represent official Docker images hosted on Docker Hub. For more information on the <code>scratch</code> image, refer to https://hub.docker.com/_/scratch/ .
ADD	Copies files into the container. We are copying our <code>vaultd</code> binary and calling it <code>vaultd</code> .
EXPOSE	Exposes the list of ports; in our case, the Vault service binds to :8080 and :8081.
ENTRYPOINT	The binary to run when the container is executed in our case, the <code>vaultd</code> binary, which will be put there by the previous call to go install.
MAINTAINER	Name and email of the person responsible for maintaining the Docker image.



For a complete list of the supported commands, consult the online Docker documentation at <https://docs.docker.com/engine/reference/builder/#dockerfile-reference>.

Building Go binaries for different architectures

Go supports cross-compilation, a mechanism by which we can build a binary on one machine (say, our Mac) targeted for a different operating system (such as Linux or Windows) and architecture. Docker containers are Linux-based; so, in order to deliver a binary that can run in that environment, we must first build one.

In a terminal, navigate to the vault folder and run the following command:

```
CGO_ENABLED=0 GOOS=linux go build -a ./cmd/vaultd/
```

We are essentially calling go build here but with a few extra bits and pieces to control the build process. CGO_ENABLED and GOOS are environment variables that go build will pay attention to, -a is a flag, and ./cmd/vaultd/ is the location of the command we want to build (in our case, the vaultd command we built in the previous chapter).

- The CGO_ENABLED=0 indicates that we do not want cgo to be enabled. Since we are not binding to any C dependencies, we can reduce the size of our build by disabling this.
- GOOS is short for Go Operating System and lets us specify which OS we are targeting, in our case, Linux. For a complete list of the available options, you can look directly in the Go source code by visiting <https://github.com/golang/go/blob/master/src/go/build/syslist.go>.

After a short while, you'll notice that a new binary has appeared, called vaultd. If you're on a non-Linux machine, you won't be able to directly execute this but don't worry; it'll run inside our Docker container just fine.

Building a Docker image

To build the image, in a terminal, navigate to Dockerfile and run the following command:

```
docker build -t vaultd
```

We are using the docker command to build the image. The final dot indicates that we want to build Dockerfile from the current directory. The -t flag specifies that we want to give our image the name of vaultd. This will allow us to refer to it by name rather than a hash that Docker will assign to it.

If this is the first time you've used Docker, and in particular the scratch base image, then it will take some time to download the required dependencies from Docker Hub depending on your Internet connection. Once that's finished, you will see output similar to the following:

```
Step 1 : FROM scratch
-->
Step 2 : MAINTAINER Your Name <your@email.address>
--> Using cache
--> a8667f8f0881
Step 3 : ADD vaultd vaultd
```

```
---> 0561c999c1e3
Removing intermediate container 4b75fde507df
Step 4 : EXPOSE 8080 8081
--> Running in 8f169f5b3b44
--> 1d7758c20b3a
Removing intermediate container 8f169f5b3b44
Step 5 : ENTRYPOINT /vaultd
--> Running in b5d55d6429be
--> b7178985dddf
Removing intermediate container b5d55d6429be
Successfully built b7178985dddf
```

For each command, a new image is created (you can see the intermediate containers being disposed of along the way) until we end up with the final image.

Since we are building our binary on our local machine and copying it into the container (with the `ADD` command), our Docker image ends up being only about 7 MB: pretty small when you consider that it contains everything it needs to run our services.

Running a Docker image locally

Now that our image is built, we can test it by running it with the following command:

```
docker run -p 6060:8080 -p 6061:8081 --name localtest --rm vaultd
```

The `docker run` command will spin up an instance of the `vaultd` image.

The `-p` flags specify a pair of ports to be exposed, the first value is the host port and the second value (following the colon) is the port within the image. In our case, we are saying that we want port 8080 to be exposed onto port 6060 and port 8081 exposed via port 6061.

We are giving the running instance a name of `localtest` with the `--name` flag, which will help us to identify it when inspecting and stopping it. The `--rm` flag indicates that we want the image to be removed once we have stopped it.

If this is successful, you will notice that the Vault service has indeed begun because it is telling us the ports to which it is bound:

```
2016/09/20 15:56:17 grpc: :8081
2016/09/20 15:56:17 http: :8080
```



These are the internal ports; remember that we have mapped these to different external ports instead. This seems confusing but ends up being very powerful, since the person responsible for spinning up the instances of the service gets to decide which ports are right for their environment, and the Vault service itself doesn't have to worry about it.

To see this running, open another terminal and use the `curl` command to access the JSON endpoint of our password hashing service:

```
curl -XPOST -d '{"password":"monkey"}' localhost:6060/hash
```

You will see something that resembles the output from the running service:

```
{"hash": "$2a$0$wk4qc74oug0kbkt/TWuRQHSG03i1ataNupbDADBwpe"}
```

Inspecting Docker processes

To see what Docker instances are running, we can use the `docker ps` command. In the terminal, type the following:

```
docker ps
```

You'll get a text table outlining the following properties:

CONTAINER ID	0b5e35dca7cc
IMAGE	vaultd
COMMAND	/bin/sh -c /go/bin/vaultd
CREATED	3 seconds ago
STATUS	Up 2 seconds
PORTS	0.0.0.0:6060->8080/tcp, 0.0.0.0:6061->8081/tcp
NAMES	localtest

The details show you a high-level overview of the image we just started. Note that the **PORTS** section shows you the mapping from external to internal.

Stopping a Docker instance

We are used to hitting *Ctrl + C* in the window running our code to stop it, but since it's running inside a container, that won't work. Instead, we need to use the `docker stop` command.

Since we gave our instance the name `localtest`, we can use this to stop it by typing this in an available terminal window:

```
docker stop localtest
```

After a few moments, you'll notice that the terminal that was running the image has now returned to the prompt.

Deploying Docker images

Now that we have contained our Vault service inside a Docker container, we are going to do some useful things with it.

The first thing we are going to do is push this to the Docker Hub so that other people may spin up their own instances or even build new images based on it.

Deploying to Docker Hub

Head over to Docker Hub at <https://hub.docker.com> and create an account by clicking on the **Log In** link in the top-right-hand corner and then clicking on **Create Account**. Of course, if you already have an account, just log in.

Now in a terminal, you are going to authenticate with this account by running Docker's `login` command:

```
docker login -u USERNAME -p PASSWORD https://index.docker.io/v1/
```



If you see an error such as `WARNING: Error loading config, permission denied`, then try the command again with the `sudo` command prefix. This goes for all of Docker commands from this point onwards, since we're using a secured configuration.

Ensure that you replace `USERNAME` and `PASSWORD` with your actual username and password of the account you just created.

If successful, you'll see, **Login Succeeded**.

Next, back in the web browser, click on **Create Repository** and create a new repository called `vault`. The actual name for this image is going to be `USERNAME/vault`, so we're going to need to rebuild the image locally to match this.



Note that for public consumption, we are calling the image `vault` rather than `vaultd`. This is a deliberate difference so that we can make sure we are dealing with the right image, but this is also a better name for users anyway.

In a terminal, build the new repository with the correct name:

```
docker build -t USERNAME/vault
```

This will build the image again, this time with the appropriate name. To deploy the image to the Docker Hub, we use Docker's push command:

```
docker push USERNAME/vault
```

After some time, the image and its dependencies will be pushed to Docker Hub:

```
f477b97e9e48: Pushed
384c907d1173: Pushed
80168d020f50: Pushed
0ceba54dae47: Pushed
4d7388e75674: Pushed
f042db76c15c: Pushing [=====>] 21.08 MB/243.6 MB
d15a527c2ee1: Pushing [=====>] 15.77 MB/134 MB
751f5d9ad6db: Pushing [=====>] 16.49 MB/122.6 MB
17587239b3df: Pushing [=====>] 17.01 MB/44.31 MB
9e63c5bce458: Pushing [=====>] 65.58 MB/125.1 MB
```

Now head over to the Docker Hub to see the details of your image, or look at an example at <https://hub.docker.com/r/matryer/vault/>.

Deploying to Digital Ocean

Digital Ocean is a cloud service provider that offers competitive prices to host virtual machines. It makes deploying and serving Docker images very easy. In this section, we are going to deploy a droplet (Digital Ocean's terminology for a single machine) that runs our dockerized Vault service in the cloud.

Specifically, following are the steps to deploy Docker images to Digital Ocean:

1. Create a droplet.
2. Gain access to it via a web-based console.
3. Pull our `USERNAME/vault` container.
4. Run the container.
5. Access our hosted Vault service remotely via the `curl` command.

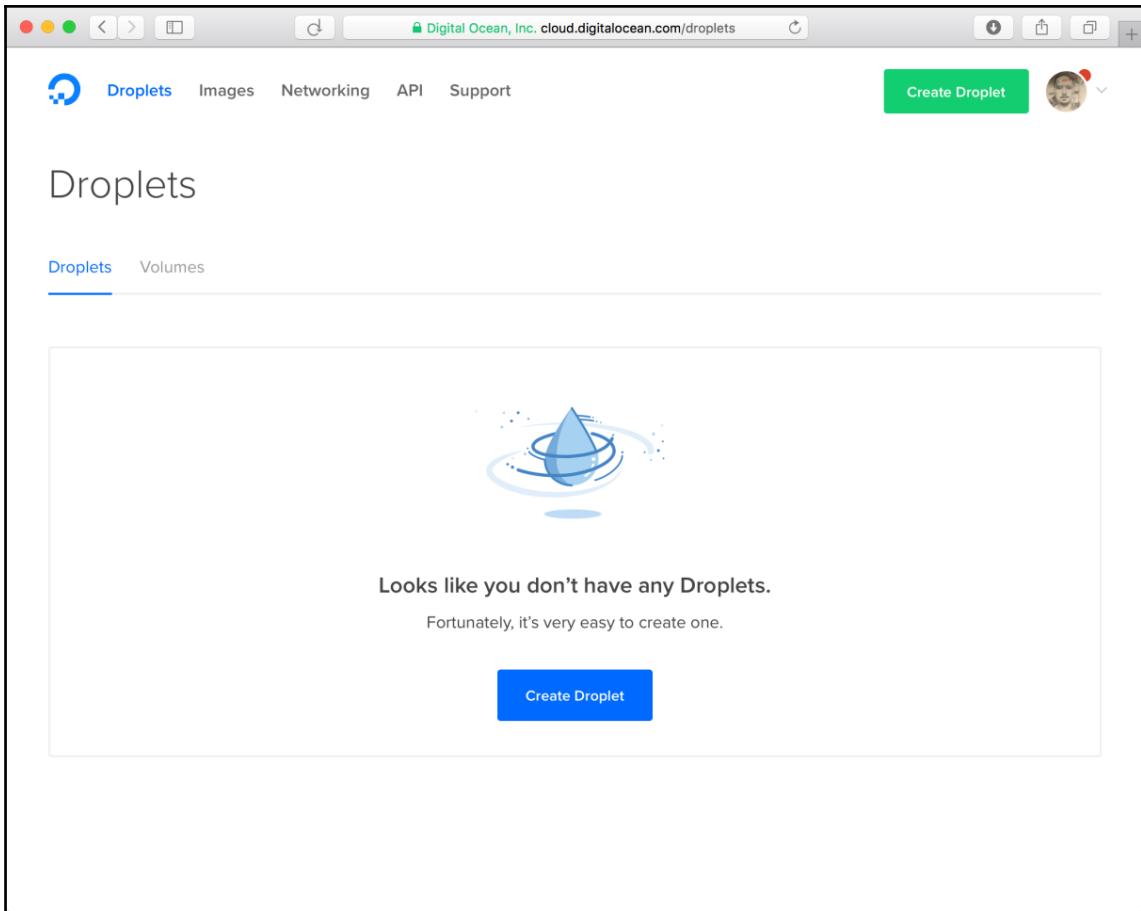
Digital Ocean is a **Platform as a Service (PaaS)** architecture, and as such, the user experience is likely to change from time to time, so the exact flow described here might not be entirely accurate by the time you come to perform these tasks. Usually, by looking around at the options, you will be able to figure out how to proceed, but screenshots have been included to help guide you.

This section also assumes that you have enabled any billing that might be required in order to create droplets.

Creating a droplet

Sign up or log in to Digital Ocean by visiting <https://www.digitalocean.com> in the browser. Ensure that you use a real e-mail address, as this is where they will send the root password for the droplet you are going to create.

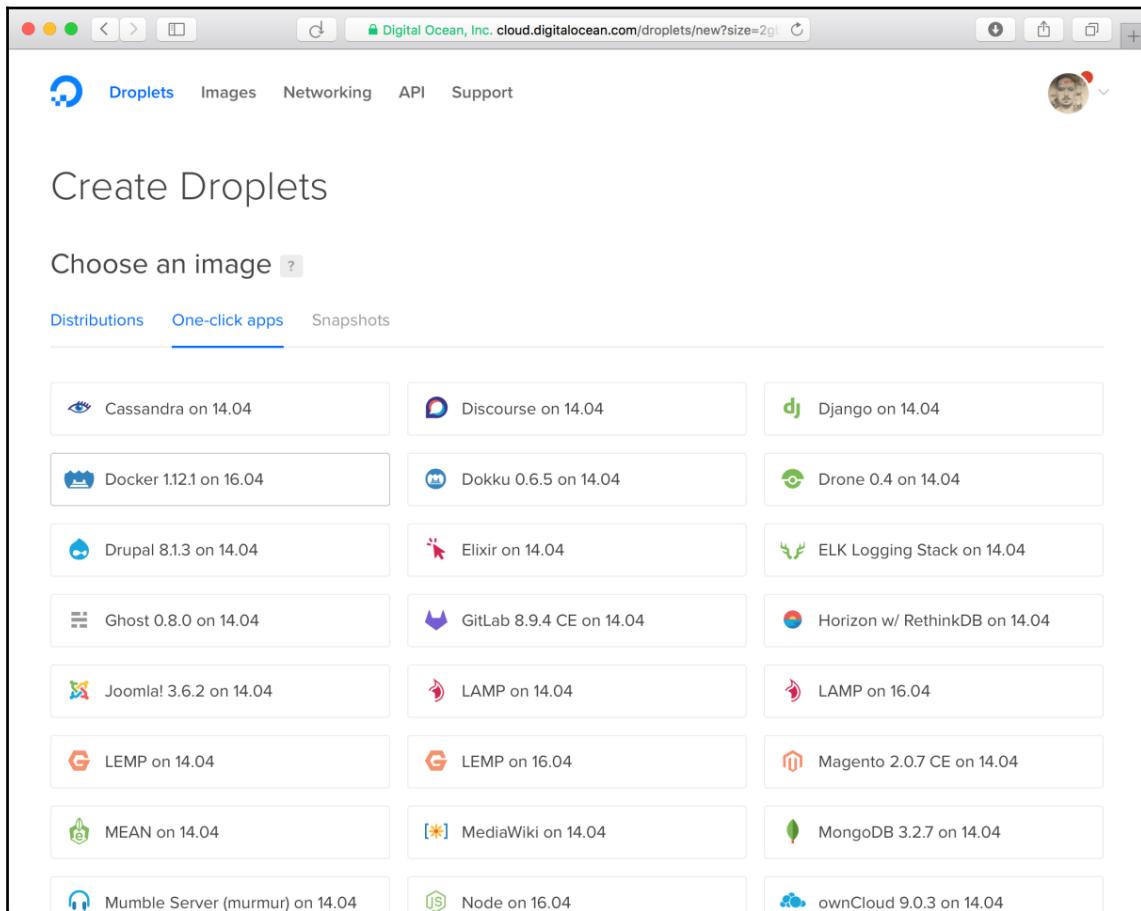
If you have no other droplets, you will be presented with a blank screen. Click on **Create Droplet**:



Inside the **One-click apps** tab, look for the latest Docker option; at the time of writing this, it is **Docker 1.12.1 on 16.04**, which means Docker version 1.12.1 is running on Ubuntu 16.04.

Scroll down the page to select the remaining options, including picking a size (the smallest size will do for now) and a location (pick the closest geographic location to you). We won't bother adding additional services (such as volumes, networking, or backups) for now just proceed with the simple droplet.

It might be a nice idea to give your droplet a meaningful hostname so that it's easy to find later, something like `vault-service-1` or similar; it doesn't really matter for now:



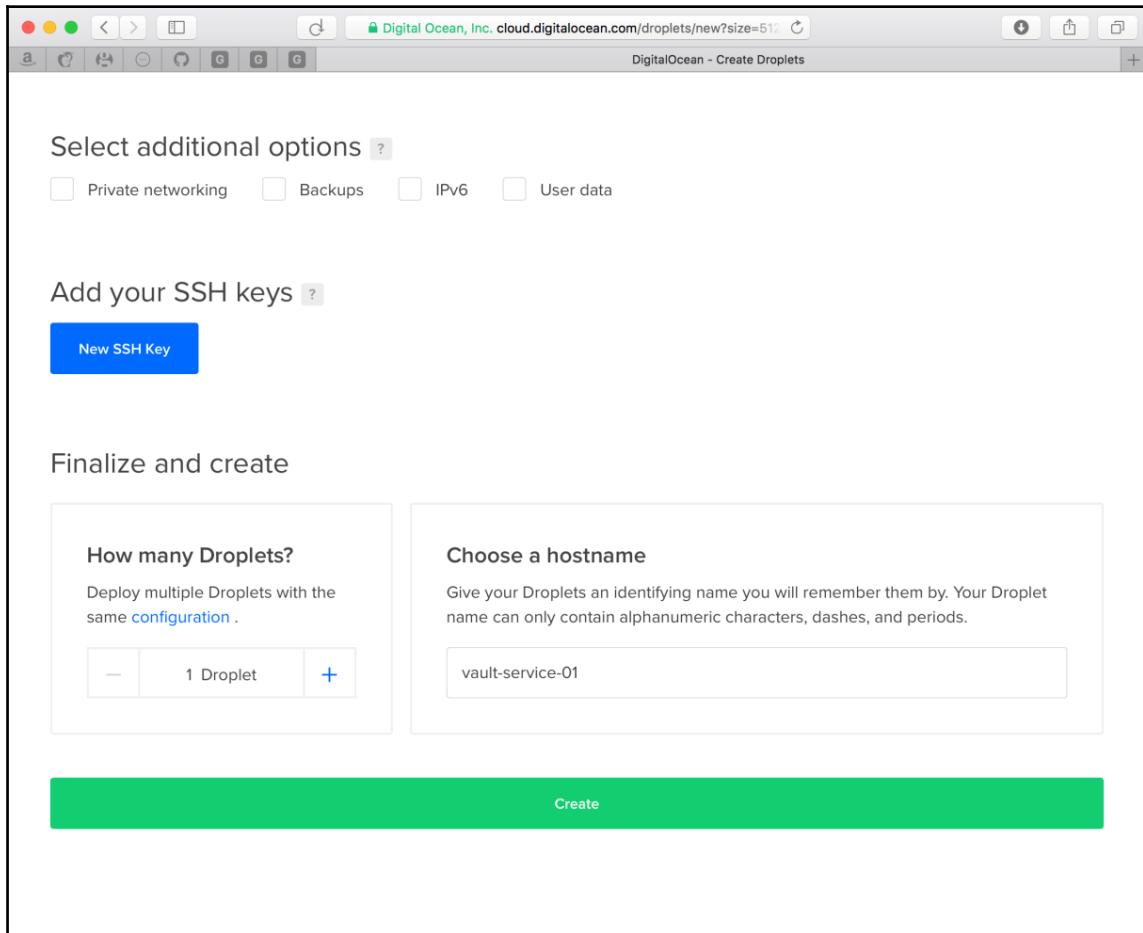
The screenshot shows the DigitalOcean 'Create Droplets' page. At the top, there are tabs for 'Distributions', 'One-click apps' (which is selected), and 'Snapshots'. Below these are several grid sections containing various pre-configured droplet options. The 'One-click apps' section contains the following items:

Cassandra on 14.04	Discourse on 14.04	Django on 14.04
Docker 1.12.1 on 16.04	Dokku 0.6.5 on 14.04	Drone 0.4 on 14.04
Drupal 8.1.3 on 14.04	Elixir on 14.04	ELK Logging Stack on 14.04
Ghost 0.8.0 on 14.04	GitLab 8.9.4 CE on 14.04	Horizon w/ RethinkDB on 14.04
Joomla! 3.6.2 on 14.04	LAMP on 14.04	LAMP on 16.04
LEMP on 14.04	LEMP on 16.04	Magento 2.0.7 CE on 14.04
MEAN on 14.04	MediaWiki on 14.04	MongoDB 3.2.7 on 14.04
Mumble Server (murmur) on 14.04	Node on 16.04	ownCloud 9.0.3 on 14.04



You can optionally add SSH keys for additional security, but for simplicity's sake, we are going to continue without it. For production, it is recommended that you always do this.

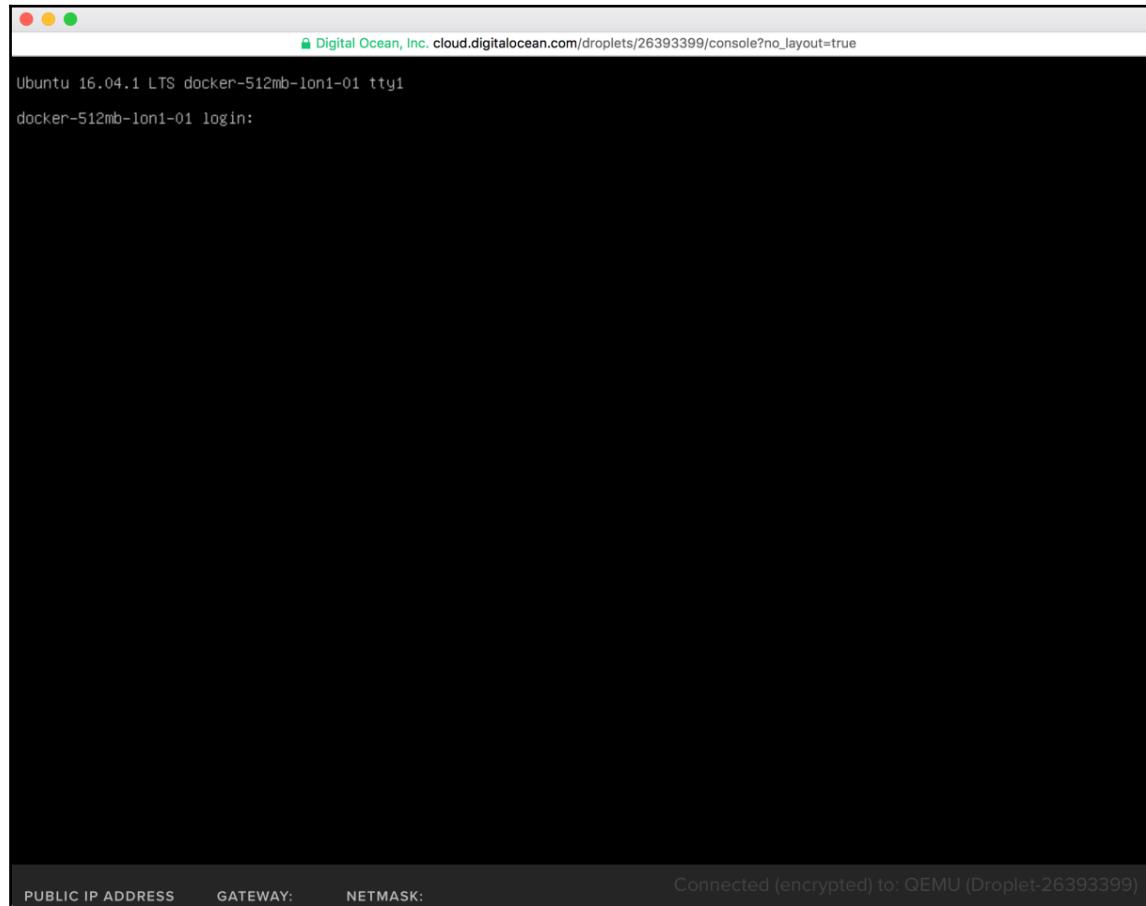
At the bottom of the page, click on **Create**:



Accessing the droplet's console

Once your droplet has been created, select it from the **Droplets** list and look for the **Console** option (it may be written as `Access console`).

After a few moments, you will be presented with a web-based terminal. This is how we will control the droplet, but first, we must log in:



Enter the login username as `root`, and check your e-mail for the root password that Digital Ocean has sent you. At the time of writing this, you cannot copy and paste this, so be ready to carefully type out a long string as accurately as you can.



The password might well be a lowercase hexadecimal string, which will help you know which characters are likely to appear. For example, everything that looks like an *O* is probably *zero*, and *I* is unlikely to be an *L* or *L*.

Once you've logged in for the first time, you'll be asked to change your password which involves typing the long generated password again! Security can be so inconvenient at times.

Pulling Docker images

Since we selected the Docker app as a starting point for our droplet, Digital Ocean has kindly configured Docker to already be running inside our instance, so we can just use the `docker` command to finish setting things up.

In the web-based terminal, pull your container with the following command, remembering to replace `USERNAME` with your Docker Hub username:

```
docker pull USERNAME/vault
```



If, for whatever reason, this isn't working for you, you can try using the Docker image placed there by the author by typing this: `docker pull matryer/vault`

Docker will go and pull down everything it needs in order to run the image we created earlier:

The screenshot shows a terminal window with the following content:

```
Docker has been preinstalled and configured per Docker's Recommendations.  
"ufw" has not been enabled, however it has been configured. To enable it,  
run "ufw enable".  
Let's Encrypt has been pre-installed for you. If you have a domain name, and  
you will be using it with this 1-Click app, please see: http://do.co/le-apache  
-----  
You can learn more about using this image here: http://do.co/docker  
-----  
To delete this message of the day: rm -rf /etc/update-motd.d/99-one-click  
The programs included with the Ubuntu system are free software;  
the exact distribution terms for each program are described in the  
individual files in /usr/share/doc/*copyright.  
Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by  
applicable law.  
root@docker-512mb-1on1-01:~# docker pull matryer/vault  
Using default tag: latest  
latest: Pulling from matryer/vault  
8ad8b3f87b37: Extracting 7.864 MB/51.37 MB  
751fe39c4d34: Download complete  
ae3b77eefc06: Download complete  
92c7f8737c98: Download complete  
bf37bbda794c: Downloading 69.2 MB/81.63 MB  
6e9d2df2553b: Download complete  
a7980310595: Waiting  
05b99b670655: Waiting  
888ae09816f2: Waiting  
9f389490be9e: Waiting  
-  
-----  
PUBLIC IP ADDRESS      GATEWAY:      NETMASK:  
Connected (encrypted) to: QEMU (Droplet-26442583)
```

Running Docker images in the cloud

Once the image and its dependencies have successfully downloaded, we will be able to run it using a the `docker run` command, this time with the `-d` flag to specify that we want it to run as a background daemon. In the web-based terminal, type the following:

```
docker run -d -p 6060:8080 -p 6061:8081 --name vault USERNAME/vault
```

This is similar to the command we ran earlier, except that this time, we are giving it the name `vault`, and we have omitted the `--rm` flag, since it is not compatible (and doesn't make sense) with the background daemon mode.

The Docker image containing our Vault service will start running and is now ready to test.

Accessing Docker images in the cloud

Now that our Docker image is running in our droplet within Digital Ocean's platform, we can start using it.

In the Digital Ocean web control panel, select **Droplets** and look for the one we just created. We need to know the IP address so that we can access the services remotely. Once you have located the IP address of the droplet, click on it to copy it.

Open a local terminal on your computer (do not use the web-based terminal) and use the `curl` command (or equivalent) to make the following request:

```
curl -XPOST -d '{"password": "Monkey"}' http://IPADDRESS:6060/hash
```

Remember to replace `IPADDRESS` with the actual IP address you copied from Digital Ocean's web control panel.

You will notice that you have successfully managed to access the JSON/HTTP endpoint of our Vault service when you get a response similar to the following:

```
{"hash": "$2a$10$eGFGRZ2zMfsXss.6CgK6/N7TsmF.6MAv6i7Km4AHC"}
```

See whether you can modify the `curl` command to validate the hash that was provided using the `/validate` endpoint.

Summary

In this chapter, we built and deployed our Vault Go application using Docker to Digital Ocean's cloud.

After installing the Docker tools, we saw how easy it was to package up our Go application into a Docker image and push it to Docker Hub. We created our Digital Ocean droplet using the helpful Docker app that they provide and controlled it via a web-based console. Once inside, we were able to pull our Docker image from the Docker Hub and run it inside our droplet.

Using the public IP of the droplet, we were then able to remotely access the Vault service's JSON/HTTP endpoint to hash and validate passwords.

12

Good Practices for a Stable Go Environment

Writing Go code is a fun and enjoyable experience, where compile-time errors rather than being a pain actually guide you to write robust, high-quality code. However, every now and then, you will encounter environmental issues that start to get in the way and break your flow. While you can usually resolve these issues after some searching and a little tweaking, setting up your development environment correctly goes a long way in reducing problems, allowing you to focus on building useful applications.

In this chapter, we are going to install Go from scratch on a new machine and discuss some of the environmental options we have and the impact they might have in the future. We will also consider how collaboration might influence some of our decisions as well as what impact open sourcing our packages might have.

Specifically, we are going to:

- Install Go on your development machine
- Learn what the `GOPATH` environment variable is for and discuss a sensible approach for its use
- Learn about the Go tools and how to use them to keep the quality of our code high
- Learn how to use a tool to automatically manage our imports
- Think about on *save* operations for our `.go` files and how we can integrate the Go tools as part of our daily development
- Look at some popular code editor options to write Go code

Installing Go

The best way to install Go is to use one of the many installers available online at <https://golang.org/dl/>. Go to the Go website and click on **Download**, and then look for the latest 1.x version for your computer. The **Featured downloads** section at the top of the page contains links to the most popular versions, so yours will probably be in that list.

The code in this book has been tested with Go 1.7, but any 1.x release will work. For future versions of Go (2.0 and higher), you may need to tweak the code as major version releases may well contain breaking changes.

Configuring Go

Go is now installed, but in order to use the tools, we must ensure that it is properly configured. To make calling the tools easier, we need to add our `go/bin` path to the `PATH` environment variable.



On Unix systems, you should add `export PATH=$PATH:/opt/go/bin` (make sure it is the path you chose when installing Go) to your `.bashrc` file. On Windows, open **System Properties** (try right-clicking on **My Computer**), and under **Advanced**, click on the **Environment Variables** button and use the UI to ensure that the `PATH` variable contains the path to your `go/bin` folder.

In a terminal (you may need to restart it for your changes to take effect), you can make sure this worked by printing the value of the `PATH` variable:

```
echo $PATH
```

Ensure that the value printed contains the correct path to your `go/bin` folder; for example, on my machine it prints as follows:

```
/usr/local/bin:/usr/bin:/bin:/opt/go/bin
```



The colons (semicolons on Windows) between the paths indicate that the `PATH` variable is actually a list of folders rather than just one folder. This indicates that each folder included will be searched when you enter commands in your terminal.

Now we can make sure the Go build we just made runs successfully:

```
go version
```

Executing the `go` command (which can be found in your `go/bin` location) like this will print out the current version for us. For example, for Go 1.77.1, you should see something similar to the following:

```
go version go1.77.1 darwin/amd64
```

Getting GOPATH right

`GOPATH` is another environment variable to a folder (such as `PATH` in the previous section) that is used to specify the location for the Go source code and the compiled binary packages. Using the `import` command in your Go programs will cause the compiler to look in the `GOPATH` location to find the packages you are referring to. When using `go get` and other commands, projects are downloaded into the `GOPATH` folder.

While the `GOPATH` location can contain a list of colon-separated folders, such as `PATH` and you can even have a different value for `GOPATH` depending on which project you are working in it is strongly recommended that you use a single `GOPATH` location for everything, and this is what we will assume you will do for the projects in this book.

Create a new folder called `go`, this time in your `Users` folder somewhere perhaps in a `Work` subfolder. This will be our `GOPATH` target and is where all the third-party code and binaries will end up as well as where we will write our Go programs and packages. Using the same technique you used when setting the `PATH` environment variable in the previous section, set the `GOPATH` variable to the new `go` folder. Let's open a terminal and use one of the newly installed commands to get a third-party package for us to use:

```
go get github.com/matryer/silk
```

Getting the `silk` library will actually cause this folder structure to be created: `$GOPATH/src/github.com/matryer/silk`. You can see that the path segments are important in how Go organizes things, which helps namespace projects and keeps them unique. For example, if you created your own package called `silk`, you wouldn't keep it in the GitHub repository of `matryer`, so the path would be different.

When we create projects in this book, you should consider a sensible `GOPATH` root for them. For example, I used `github.com/matryer/goblueprints`, and if you were to `go get` that, you would actually get a complete copy of all the source code for this book in your `GOPATH` folder!

Go tools

An early decision made by the Go core team was that all Go code should look familiar and obvious to everybody who speaks Go rather than each code base requiring additional learning in order for new programmers to understand it or work on it. This is an especially sensible approach when you consider open source projects, some of which have hundreds of contributors coming and going all the time.

There is a range of tools that can assist us in achieving the high standards set by the Go core team, and we will look at some of the tools in action in this section.

In your `GOPATH` location, create a new folder called `tooling` and create a new `main.go` file containing the following code verbatim:

```
package main
import (
    "fmt"
)
func main() {
    return
    var name string
    name = "Mat"
    fmt.Println("Hello ", name)
}
```

The tight spaces and lack of indentation are deliberate as we are going to look at a very cool utility that comes with Go.

In a terminal, navigate to your new folder and run this:

```
go fmt -w
```



At Gophercon 2014 in Denver, Colorado, most people learned that rather than pronouncing this little triad as *format* or *f, m, t*, it is actually pronounced as a word. Try saying it to yourself now: *fhumt*; it seems that computer programmers aren't weird enough without speaking an alien language to each other too!

You will notice that this little tool has actually tweaked our code file to ensure that the layout (or format) of our program matches Go standards. The new version is much easier to read:

```
package main
import (
    "fmt"
)
```

```
func main() {
    return
    var name string
    name = "Mat"
    fmt.Println("Hello ", name)
}
```

The `go fmt` command cares about indentation, code blocks, unnecessary whitespace, unnecessary extra line feeds, and more. Formatting your code in this way is a great practice to ensure that your Go code looks like all other Go code.

Next, we are going to vet our program to make sure that we haven't made any mistakes or decisions that might be confusing to our users; we can do this automatically with another great tool that we get for free:

```
go vet
```

The output for our little program points out an obvious and glaring mistake:

```
main.go:10: unreachable code
exit status 1
```

We are calling `return` at the top of our function and then trying to do other things. The `go vet` tool has noticed this and points out that we have unreachable code in our file.

It isn't just silly mistakes like this that `go vet` will catch; it will also look for subtler aspects of your program that will guide you toward writing the best Go code you can. For an up-to-date list of what the `vet` tool will report on, check out the documentation at <https://golang.org/cmd/vet/>.

The final tool we will play with is called `goimports`, and it was written by Brad Fitzpatrick to automatically fix (add or remove) `import` statements for Go files. It is an error in Go to import a package and not use it, and obviously, trying to use a package without importing it won't work either. The `goimports` tool will automatically rewrite our `import` statement based on the contents of our code file. First, let's install `goimports` with this familiar command:

```
go get golang.org/x/tools/cmd/goimports
```

Update your program to import some packages that we are not going to use and remove the `fmt` package:

```
import (
    "net/http"
    "sync"
)
```

When we try to run our program by calling `go run main.go`, we will see that we get some errors:

```
./main.go:4: imported and not used: "net/http"
./main.go:5: imported and not used: "sync"
./main.go:13: undefined: fmt
```

These errors tell us that we have imported packages that we are not using and missing the `fmt` package and that in order to continue, we need to make corrections. This is where `goimports` comes in:

```
goimports -w *.go
```

We are calling the `goimports` command with the `-w` write flag, which will save us the task of making corrections to all files ending with `.go`.

Have a look at your `main.go` file now, and note that the `net/http` and `sync` packages have been removed and the `fmt` package has been put back in.

You could argue that switching to a terminal to run these commands takes more time than just doing it manually, and you would probably be right in most cases, which is why it is highly recommended that you integrate the Go tools with your text editor.

Cleaning up, building, and running tests on save

Since the Go core team has provided us with such great tools as `fmt`, `vet`, `test`, and `goimports`, we are going to look at a development practice that has proven to be extremely useful. Whenever we save a `.go` file, we want to perform the following tasks automatically:

1. Use `goimports` and `fmt` to fix our imports and format the code.
2. Vet the code for any faux pas and tell us immediately.
3. Attempt to build the current package and output any build errors.
4. If the build is successful, run the tests for the package and output any failures.

Because Go code compiles so quickly (Rob Pike once actually said that it doesn't build quickly, but it's just not slow like everything else), we can comfortably build entire packages every time we save a file. This is also true for running tests to help us if we are developing in a TDD style, and the experience is great. Every time we make changes to our code, we can immediately see whether we have broken something or had an unexpected impact on some other part of our project. We'll never see package import errors again because our `import` statement will have been fixed for us, and our code will be correctly formatted right in front of our eyes.

Some editors are likely to not support running code in response to specific events, such as saving a file, which leaves you with two options: you can either switch to a better editor, or you can write your own script file that runs in response to filesystem changes. The latter solution is out of the scope of this book; instead, we will focus on how to implement this functionality in a couple of popular editor codes.

Integrated developer environments

The **Integrated Developer Environments (IDEs)** are essentially text editors with additional features that make writing code and building software easier. Text with special meaning, such as string literals, types, function names, and so on are often colored differently by syntax highlighting, or you may get autocomplete options as you're typing. Some editors even point out errors in your code before you've executed it.

There are many options to choose from, and mostly, it comes down to personal preference, but we will look at some of the more popular choices as well as how to set them up to build Go projects.

The most popular editors include the following:

- Sublime Text 3
- Visual Studio Code
- Atom
- Vim (with vim-go)

You can see a complete curated list of options at <https://github.com/golang/go/wiki/IDEsAndTextEditorPlugins>.

In this section, we are going to explore Sublime Text 3 and Visual Studio Code.

Sublime Text 3

Sublime Text 3 is an excellent editor to write Go code that runs on OS X, Linux, and Windows and has an extremely powerful expansion model, which makes it easy to customize and extend. You can download Sublime Text from <http://www.sublimetext.com/> and trial-use it for free before deciding whether you want to buy it or not.

Thanks to **DisposaBoy** (refer to <https://github.com/DisposaBoy>), there is already a Sublime expansion package for Go, which actually gives us a wealth of features and power that a lot of Go programmers actually miss out on. We are going to install this **GoSublime** package and then build upon it to add our desired on-save functionality.

Before we can install **GoSublime**, we need to install Package Control into Sublime Text. Head over to <https://sublime.wbond.net/> and click on the **Installation** link for instructions on how to install Package Control. At the time of writing this, it's simply a case of copying the single, albeit long, line command and pasting it into the Sublime console, which can be opened by navigating to **View | Show Console** from the menu.

Once this is complete, press *shift + command + P* and type **Package Control: Install Package** and press *return* when you have selected the option. After a short delay (where Package Control is updating its listings), a box will appear, allowing you to search for and install **GoSublime** just by typing it in, selecting it, and pressing *return*. If all is well, **GoSublime** will be installed and writing Go code will just become an order of magnitude easier.



Now that you have **GoSublime** installed, you can open a short help file containing the details of the package by pressing *command + ., command + 2* (the command key and period at the same time, followed by the command key and number 2).

For some additional help while saving, press *command + ., command + 5* to open the **GoSublime** settings and add the following entry to the object:

```
"on_save": [
  {
    "cmd": "gs9o_open",
    "args": {
      "run": ["sh", "go build . errors && go test -i && go test &&
              go vet && golint"],
      "focus_view": false
    }
  }
]
```



Note that the settings file is actually a JSON object, so ensure that you add the `on_save` property without corrupting the file. For example, if you have properties before and after, ensure the appropriate commas are in place.

The preceding setting will tell Sublime Text to build the code looking for errors, install test dependencies, run tests, and vet the code whenever we save the file. Save the settings file (don't close it just yet), and let's see this in action.

Navigate to **Choose File | Open...** from the menu and select a folder to open for now, let's open our `tooling` folder. The simple user interface of Sublime Text makes it clear that we only have one file in our project right now: `main.go`. Click on the file and add some extra linefeeds, and add and remove some indenting. Then, navigate to **File | Save** from the menu, or press `command + S`. Note that the code is immediately cleaned up, and provided that you haven't removed the oddly placed return statement from `main.go`, you will notice that the console has appeared and is reporting the issue thanks to go vet:

```
main.go:8: unreachable code
```

Holding down `command + shift` and double-clicking on the unreachable code line in the console will open the file and jump the cursor to the right line in question. You can see how helpful this feature is going to be as you continue to write Go code.

If you add an unwanted import to the file, you will notice that on using `on_save`, you are told about the problem, but it wasn't automatically fixed. This is because we have another tweak to make. In the same settings file as the one you added the `on_save` property to, add the following property:

```
"fmt_cmd": ["goimports"]
```

This tells GoSublime to use the `goimports` command instead of `go fmt`. Save this file again, and head back to `main.go`. Add `net/http` to the imports again, remove `fmt` import, and save the file. Note that the unused package was removed, and `fmt` was put back again.

Visual Studio Code

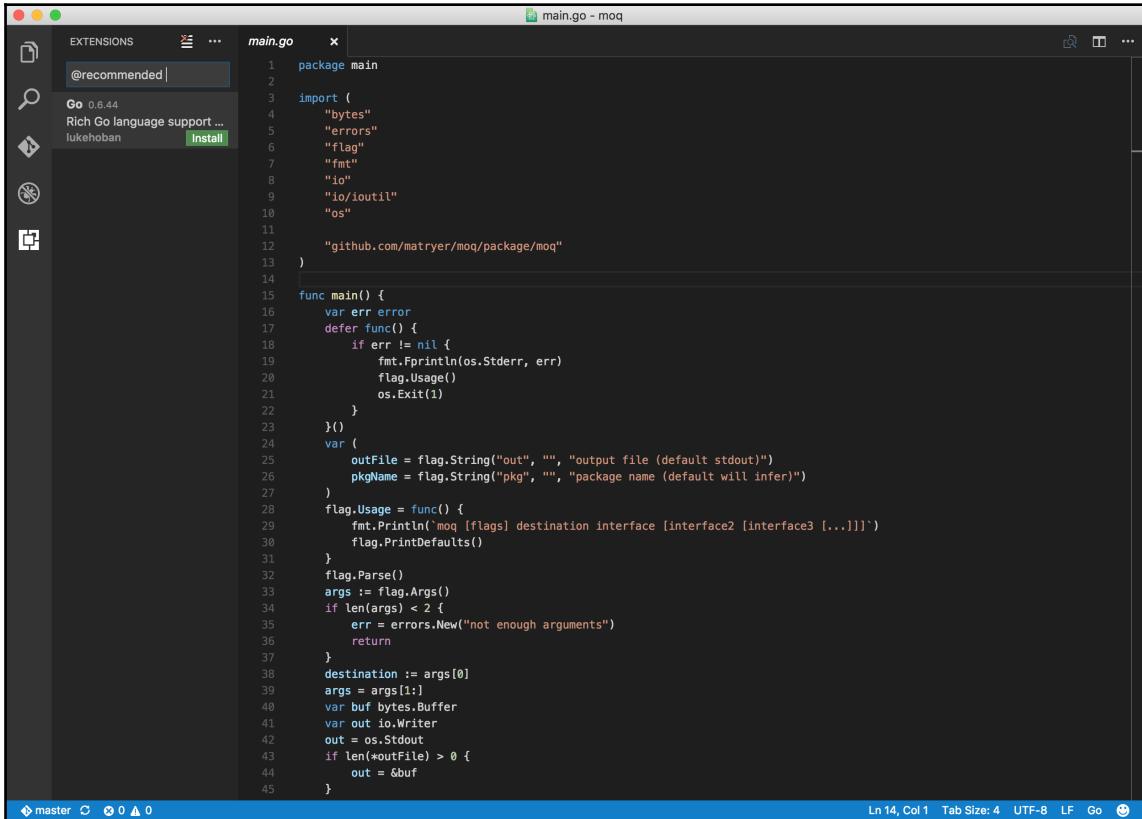
A surprise entry in the running for best Go IDE is Microsoft's Visual Studio Code, available for free at <https://code.visualstudio.com>.

Once you've downloaded it from the website, open a Go file (any file with a .go extension) and note that Visual Studio Code asks whether you'd like to install the recommended plugins to make working with Go files easier:

```
main.go - moq
EXPLORER Info It is recommended to install the 'lukehoban.Go' extension.
SHOW RECOMMENDATIONS Don't show again CLOSE
OPEN EDITORS
  main.go
MOQ
  example
  generate
  package
  .gitignore
  .travis.yml
LICENSE
  main.go
moq-logo.png
moq-logo-small.png
README.md

1 package main
2
3 import (
4     "bytes"
5     "errors"
6     "flag"
7     "fmt"
8     "io"
9     "io/ioutil"
10    "os"
11
12    "github.com/matryer/moq/package/moq"
13)
14
15 func main() {
16     var err error
17     defer func() {
18         if err != nil {
19             fmt.Fprintln(os.Stderr, err)
20             flag.Usage()
21             os.Exit(1)
22         }
23     }()
24     var (
25         outFile = flag.String("out", "", "output file (default stdout)")
26         pkgName = flag.String("pkg", "", "package name (default will infer)")
27     )
28     flag.Usage = func() {
29         fmt.Println(`moq [flags] destination interface [interface2 [interface3 [...]]]`)
30         flag.PrintDefaults()
31     }
32     flag.Parse()
33     args := flag.Args()
34     if len(args) < 2 {
35         err = errors.New("not enough arguments")
36         return
37     }
38     destination := args[0]
39     args = args[1:]
40     var buf bytes.Buffer
41     var out io.Writer
42     out = os.Stdout
43     if len(*outFile) > 0 {
44         out = &buf
45     }
}
```

Click on **Show Recommendations** and click on **Install** next to the suggested Go plugin:



The screenshot shows the Visual Studio Code interface with the Go extension installed. In the left sidebar, under 'EXTENSIONS', there is a recommendation for 'Go 0.6.44' by 'lukehoban'. A green 'Install' button is visible next to the extension name. The main editor tab is titled 'main.go' and contains Go code for a command-line application. The status bar at the bottom shows 'Ln 14, Col 1 Tab Size: 4 UTF-8 LF Go'.

```
1 package main
2
3 import (
4     "bytes"
5     "errors"
6     "flag"
7     "fmt"
8     "io"
9     "io/ioutil"
10    "os"
11
12    "github.com/matryer/moq/package/moq"
13 )
14
15 func main() {
16     var err error
17     defer func() {
18         if err != nil {
19             fmt.Fprintln(os.Stderr, err)
20             flag.Usage()
21             os.Exit(1)
22         }
23     }()
24     var (
25         outFile = flag.String("out", "", "output file (default stdout)")
26         pkgName = flag.String("pkg", "", "package name (default will infer)")
27     )
28     flag.Usage = func() {
29         fmt.Println(`moq [flags] destination interface [interface2 [interface3 [...]]]`)
30         flag.PrintDefaults()
31     }
32     flag.Parse()
33     args := flag.Args()
34     if len(args) < 2 {
35         err = errors.New("not enough arguments")
36         return
37     }
38     destination := args[0]
39     args = args[1:]
40     var buf bytes.Buffer
41     var out io.Writer
42     out = os.Stdout
43     if len(*outFile) > 0 {
44         out = &buf
45     }
```

It may ask you to restart Visual Studio Code to enable the plugin, and it may also ask you to install some additional commands:

The screenshot shows the Visual Studio Code interface with the Go extension installed. The Explorer sidebar lists files: main.go, MOQ, example, generate, package, .gitignore, .travis.yml, LICENSE, main.go, mod-logo.png, mod-logo-small.png, and README.md. The main editor window displays the following Go code:

```
3 import (
4     "bytes"
5     "errors"
6     "flag"
7     "fmt"
8     "io"
9     "io/ioutil"
10    "os"
11
12    "github.com/matryer/moq/package/moq"
13)
14
15 func main() {
16     var err error
17     defer func() {
18         if err != nil {
19             fmt.Fprintln(os.Stderr, err)
20             flag.Usage()
21             os.Exit(1)
22         }
23     }()
24     var (
25         outFile = flag.String("out", "", "output file (default stdout)")
26         pkgName = flag.String("pkg", "", "package name (default will infer)")
27     )
28     flag.Usage = func() {
29         fmt.Println(`moq [flags] destination interface [interface2 [interface3 [...]]]`)
30         flag.PrintDefaults()
31     }
32     flag.Parse()
33     args := flag.Args()
34     if len(args) < 2 {
35         err = errors.New("not enough arguments")
36         return
37     }
38     destination := args[0]
39     args = args[1:]
40     var buf bytes.Buffer
41     var out io.Writer
42     out = os.Stdout
43     if len(*outFile) > 0 {
44         out = &buf
45     }

```

The status bar at the bottom shows: master, Ln 22, Col 13, Tab Size: 4, UTF-8, LF, Go, Analysis Tools Missing.

Click on **Install All** to install all the dependencies, being sure to wait for the previous installation process to finish before initiating others. After a short while, you will notice that a few tools were installed.

Write some messy code (or copy and paste some from <https://github.com/matryer/goblueprints/blob/master/appendixA/messycode/main.go>) into Visual Studio Code and hit save. You will notice that the imports were fixed and the code was nicely formatted as per the Go standard.

There are many more features that you can make use of, but we won't dig into them further here.

Summary

In this appendix, we installed Go and are now ready to start building real projects. We learned about the `GOPATH` environment variable and discovered a common practice of keeping one value for all projects. This approach dramatically simplifies working on Go projects, where you are likely to continue to encounter tricky failures otherwise.

We discovered how the Go toolset can really help us produce high-quality, community-standards-compliant code that any other programmer could pick up and work on with little to no additional learning. And more importantly, we looked at how automating the use of these tools means we can truly get down to the business of writing applications and solving problems, which is all developers really want to do.

We looked at a couple of options for code editors or IDEs and saw how easy it was to add plugins or extensions that help writing Go code easier.

Index

A

API

serving, with one function 171

Application Programming Interface (API)

28

apps, running with multiple modules

about 290

testing, admin console used 291

testing, locally 290

apps

deploying, with multiple modules 292

arguments, parsing

about 231

paths, adding 233

paths, listing 232

paths, removing 233

string representations, for custom types 232

authorization providers

informing, about application 51, 52

authorization, with Twitter

about 136

connection, extracting 137

environment variables, reading 138, 139

Available program

118, 119, 121

avatar implementation, for local files

about 87

different file types, supporting 89

avatar implementations

combining 98, 99

avatar picture

avatar implementation, for local files 87

code, optimizing 90

code, refactoring 90

images, serving 86

upload form 83, 84

upload, handling 84, 86

uploading 81

user identification 82

avatar URL process, Gravatar

auth service 74

avatar implementation 74, 76

avatar implementation, using 76, 78

Gravatar implementation 78, 79

avatars, OAuth2 server

about 66

avatar URL, obtaining 66

avatar URL, transmitting 67

avatar, adding to user interface 68

enhancing 71, 72

logging out 69, 70

B

backup package

about 220

backup, initiating 227

changes, checking for 226

hardcoding 228

interfaces, considering 220

interfaces, testing 221, 223

issues 224, 226

Base64-encode

57

bcrypt

passwords, hashing with 304

passwords, validating with 304

Big Huge Thesaurus

URL 112

BSON (Binary JSON)

156

C

chat application

simple web server 10

chat room

client, modeling 16, 17

concurrency programming, idiomatic Go used

19, 22
creating 23
helper functions, used for removing complexity 22
modeling 15, 16, 19
turning, into HTTP handler 20
using 23

cloud
Docker images, accessing in 348
Docker images, running in 348

Coda Hale blog post
reference 304

code optimization, avatar
about 90
concrete types, replacing with interfaces 91
existing implementations, fixing 94, 95
global variables, versus fields 95, 96
interfaces, changing in test-driven way 92, 93
new design, implementing 96, 97
testing 97, 98
tidying up 97

command server
gRPC server, running 319

commands, Dockerfile
ADD 335
ENTRYPOINT 335
EXPOSE 335
FROM 335

constructors, in Go 303

Coolify
about 109
domain suggestions 117, 118
environment variables, using for configuration 113
web API, consuming 113, 116, 117
working 109, 111

CORS (Cross-origin resource sharing) technique 166, 213

D

daemon backup tool
about 235
data, caching 237
duplicated structures 237
filedb records, updating 239

infinite loops 238
data access 255
data operations, exposing over HTTP
about 277
context, in Google App Engine 282
HTTP routing in Go 281
key strings, decoding 283, 285
optional features, with type assertions 277, 278
path parameters, parsing 279, 280, 281

data sharing
between handlers 163
context keys 163, 164, 165

data
denormalizing 253, 254, 255
putting, into Google Cloud Datastore 257, 258
reading, from Google Cloud Datastore 259
representing, in code 197, 198, 199

database design 131

dependencies
injecting 167

Digital Ocean
about 341
Docker images, deploying to 341
reference 341

Docker documentation
reference 335

Docker Hub
Docker images, deploying to 339, 340
reference 339

Docker image
building 336
running, locally 337

Docker images
accessing, in cloud 348
deploying 339
deploying, to Digital Ocean 341
deploying, to Docker Hub 339, 340
pulling 346, 347
running, in cloud 348

Docker instance
stopping 339

Docker processes
inspecting 338

Docker tools
installing 334

Docker
about 333
Go binaries, building for different architectures 335, 336
reference, for project home page 334
reference, for source code 333
using, locally 334

Dockerfile
about 334
commands 335

Domainify
about 107
building 108
running 108

droplet's console
accessing 344, 345, 346

droplet
creating 341, 342, 343, 344

DRY (Don't Repeat Yourself) 167

E

editors
Sublime Text 3 357
Visual Studio Code 359

endpoints, in Go kit 307

endpoints
API, testing with curl 180
handling 173
making, for service methods 308
many operations, with single handler 174
tags, used for adding metadata to structs 174
with dynamic paths 47, 48, 49
wrapping, into Service implementation 309

entities 255

enumerator
testing 205, 208

environment
installing 131
NSQ 131, 132
starting 134

errors, in Go kit
levels 309

external logging in
implementing 52, 53, 55
messages, augmenting with additional data 59,

60, 62, 63
response from provider, handling 56, 58
user data, presenting 58, 59

F

filesystem backup
solution design 219

G

Go code
generating 300, 301

Go kit
about 295
reference 294

Go program
writing 123, 126

Go structs
public views 200, 201

Go
about 9
configuring 351
installation 351
installation link 351
reference 354
tools 353, 354

Google App Engine SDK, for Go
about 244
app.yaml file 246, 247
application, building 246
application, creating 245
modules 250, 251
simple applications, deploying to Google App Engine 249, 250
simple applications, running locally 247, 248, 249

Google App Engine users
about 259, 260, 261
denormalized data, embedding 261, 262

Google Cloud Datastore
about 252
data, putting into 257, 258
data, reading from 259
keys 256

Google Places API key 203

Google Places API

querying 209
GoPATH 352
Gravatar
 avatar URL process, abstracting 73, 74
 implementing 73
gRPC (Google's Remote Procedure Call) 296
gRPC client
 arguments, passing in CLI 324
 building 321
 CLI tool, for consuming service 323
 good line of sight, maintaining 325
 tools, installing from Go source code 326
gRPC server, in Go kit
 about 312
 protocol buffer types, translating to types 313

H

handler function wrappers
 using 173
handler functions
 cross-origin resource sharing 166
 wrapping 165
handlers 42
horizontal scaling 128
HTML and JavaScript chat client
 building 23, 25
HTTP server, in Go kit 311

I

Integrated Developer Environments (IDEs)
 about 356
 options, reference 356
interfaces 29, 30

K

key strings, decoding
 anonymous structs, for request data 286
 query parameters used 285
 self-similar code, writing 287
 validation method, that return error 288
keys, in Google Cloud Datastore 256

L

log package

reference 258

M

Meander project, random recommendations web service
 building 194, 195
 project design specifics 195, 196
method calls
 modeling, with requests 305
 modeling, with responses 305
micro-services 294
modules, in Google App Engine
 routing, with dispatch.yaml 252
 specifying 251
MongoDB driver, for Go 134
MongoDB
 about 133
 download link 133

N

NoOps 243
NSQ driver, for Go 133
NSQ
 about 131
 installing 132
 running 132

O

OAuth2 server
 avatars 66
OAuth2
 about 50
 flow 50
 open source OAuth2 packages 50
operations, with single handler
 about 174
 CORS support 180
 poll, creating 178
 poll, deleting 179
 polls, reading 176, 178

P

Package Control
 installation link 357

package
writing, TDD used 28
password
hashing, with bcrypt 304
validating, with bcrypt 304
pipe design
for command-line tools 102
Platform as a Service (PaaS) 341
protocol buffers language
about 298, 299, 300
reference 300
protocol buffers
about 297, 298
installing 298

Q

querying, in Google Cloud Datastore 267, 268

R

random recommendations web service
Meander project, building 193
random recommendations
API, testing 214
building 210
CORS 213
enumerators, in Go 203, 204
generating 201
Google Places API key 203
Google Places API, querying 209, 210
handlers, using query parameters 212
rate limiting, with service middleware
about 327
graceful rate limiting 331
middleware, in Go kit 328, 329
rate limiter, testing manually 330
redirection metacharacters 102
request
method calls, modeling with 305
responding 167
response helpers 278
responses
method calls, modeling with 305
RESTful API design 162

S

server command
creating 315, 316
Go kit endpoints, using 318
HTTP server, running 318
main function, preventing from terminating 320
service, consuming over HTTP 320
Service implementation
endpoints, wrapping into 309
service methods
endpoints, making for 308
simple programs
Available 103
composing 122, 123
Coolify 103
Domainify 102
key features 102
Sprinkle 102
Synonyms 103
simple web server
about 10
Go programs, building 15
Go programs, executing 15
views, separating from logic with templates 12, 13
SOA (service-oriented architecture) 294
social sign-in page
building 45
solution design
about 219
project structure 219
testing 240
solution
running 158, 159
Sprinkle program
about 103
working 103, 105, 106
Sublime Text 3
about 357, 358
reference 357
Synonyms 112
system design
about 129, 130
database design 130

T

templates
 using 25, 26, 27
Test-driven Development (TDD) 28, 74
tests
 building, on save 355
 cleaning up, on save 355
 running, on save 355
Token Bucket-based rate limiter
 reference 328
Top-level Domain (TLD) 107
tracing 28
tracing code
 clean package APIs 39
 interface, implementing 34, 35
 interfaces 29, 30
 new trace package 36
 package, writing with TDD 28, 29
 tracing, making optional 38, 39
 unexported types, returning to users 35
 unit tests 30, 31
 writing 28
transactions, in Google Cloud Datastore
 about 262, 263
 early abstraction, avoiding 267
 used, for maintaining counters 263, 264, 266
Transport Layer Security (TLS) 320
twittervotes program 135

U

unit testing
 red-green testing 34
unit tests
 about 30, 31
 red-green testing 32, 33
user account
 handlers 42
user command-line tool
 about 229
 arguments, parsing 231
 small data, persisting 230
 using 234

V

Vault service
 building 301
 implementing 302
 tests, starting 302
vertical scaling 128
views separating, from logic
 custom handlers, using 14
 templates used 12, 13
 templates, compiling 14
Visual Studio Code
 about 359, 360, 361
 reference 359, 361
Vote structure
 about 269
 indexing 270, 271, 273
vote, casting
 about 273
 line of sight, in code 274, 275, 277
 parents, accessing via datastore.Key 274
votes, counting
 about 151
 database update, maintaining 155, 156
 database, connecting 152
 messages, consuming in NSQ 153, 154
 responding, to Ctrl + C 157
votes, reading from Twitter
 about 135
 authorization, with Twitter 136, 137
 MongoDB, reading from 140
 NSQ, publishing to 146
 programs, starting 148, 149
 programs, stopping 148, 150
 reading, from Twitter 142
 signal channels 144, 145, 146
 testing 150, 151

W

web application
 testing 216
web client, consuming API
 about 182
 new poll, creating 185
 poll details, displaying 186, 188

WHOIS server 103

Bibliography

This learning path has been prepared for you to help you build production-ready solutions in Go using cutting-edge technology and techniques. It comprises of the following Packt products:

- *Learning Go programming, Vladimir Vivien*
- *Go Design Patterns, Mario Castro Contreras*
- *Go Programming Blueprints - Second Edition, Mat Ryer*



Thank you for buying **Go: Design Patterns for Real-World Projects**

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

Please check www.PacktPub.com for information on our titles