Go as an Instrument of Continuation

From Local Workflows to Concurrent Systems, Containers, and Governed Services

Flyxion
Independent Researcher

October 2026

Contents

Preface **ix**

I Local Computation **1**

1 A Program Is a Controlled Transition **3**
1.1 The Case for a Compiled Tool . . . 3
1.2 The Smallest Program . . . 4
1.3 Computation as a Transition System . . . 4
1.4 Admissibility . . . 6
1.5 Admissibility Without Overflow . . . 7
1.6 From Transitions to Workflows . . . 10

2 Values, Types, and Records **13**
2.1 Structs, Copies, and Zero Values . . . 13
2.2 Named Types as Vocabularies . . . 15
2.3 Methods and the Package Boundary . . . 16
2.4 Packages, Modules, and the Internal Directory . . . 16
2.5 Two Types for Two States of Knowledge . . . 17

3 Traversing, Reading, and Hashing **19**
3.1 Walking a Directory Tree . . . 19
3.2 Reading Without Loading . . . 21
3.3 What a Digest Establishes . . . 22
3.4 Testing Against the Filesystem . . . 22

4 Errors as Values **27**
4.1 The Error Interface . . . 27
4.2 Sentinel Errors and Inspection . . . 27
4.3 Context and the Cost of Silence . . . 28
4.4 Partial Failure as Data . . . 29
4.5 Errors and Panics . . . 29

5 Running Programs Without a Shell **31**
5.1 The Argument Vector . . . 31
5.2 Four Endings . . . 33
5.3 Time and Cancellation . . . 34
5.4 A Shell Pipeline and Its Go Counterpart . . . 35

6 A Ledger of Actions **37**
6.1 What a Ledger Is For . . . 37
6.2 The Record and Its Digest . . . 37
6.3 What the Chain Establishes . . . 43
6.4 The Command and a Recorded Session . . . 46
6.5 Where Part I Ends . . . 47

II Governed Workflows **49**

7 Interfaces and Composition **51**
7.1 Why Policy Is a Separate Thing . . . 51
7.2 Verdicts as an Ordered Vocabulary . . . 54
7.3 Functions with Methods . . . 54
7.4 Composition and What It Preserves . . . 55

8 Inscription and Validation **57**
8.1 Three Questions That Are Easy to Merge . . . 57
8.2 Inscription . . . 59
8.3 Validation . . . 60
8.4 Validity Does Not Mention Policy . . . 65
8.5 The Zero Value, Again . . . 66

9 Admission, Projection, and Refusal **67**
9.1 Admission . . . 67
9.2 Seven Dispositions, Five Verdicts . . . 67
9.3 Projection and Activation . . . 68
9.4 What the Tests Check . . . 73
9.5 Recording Refusals . . . 73
9.6 Where Part II Ends . . . 75

III Concurrent Execution **77**

10 Goroutines and Channels **79**
10.1 What a Goroutine Is . . . 79
10.2 Why `golab` Needs Them . . . 79
10.3 Reading the Pool . . . 82
10.4 Why the Caller Can Read the Results . . . 83
10.5 A Race, Observed . . . 83

10.6 Goroutines Are Not Services 85

11 Contexts, Select, and Cancellation **91**
11.1 The Problem of Stopping 91
11.2 The Select Statement 91
11.3 The Pool's Cancellation Contract 92
11.4 Cancellation in the Command 96

12 Shared State and the Single Writer **99**
12.1 The Fork That Part I Predicted 99
12.2 Two Ways to Serialize 100
12.3 The Writer 100
12.4 What the Tests Establish 108
12.5 Costs and Where Part III Ends 108

IV Memory and Reconstruction **111**

13 Encoding, Buffering, and Atomic Replacement **113**
13.1 Data That Outlives a Process 113
13.2 Encoding as a Round Trip 113
13.3 Buffering and a Limit That Was There All Along 114
13.4 Replacing a File Whole 114
13.5 What the Tests Establish 117

14 Derived Data and the Index **123**
14.1 Authority and Convenience 123
14.2 The Package 123
14.3 Reconstruction Equals Incremental Update 129
14.4 What the Index Does Not Protect Against 134

15 Reconstruction and Its Limits **137**
15.1 Two Commands 137
15.2 A Recorded Session 137
15.3 What Was Not Shown 139
15.4 Limits of the Analogy 140

V Distributed Systems **143**

16 HTTP Servers and Handlers **145**
16.1 From a Command to a Service 145
16.2 Reading and Writing a Request 152
16.3 One Goroutine per Request 152
16.4 Timeouts and Shutdown 153
16.5 Testing Without a Network 153

17 Routing, Adjudication, and Admission **155**
17.1 Three Questions Again . . . 155
17.2 Routing Decides From the Request Line . . . 155
17.3 Adjudication Is a Function of Two Values . . . 157
17.4 Routing Comes First . . . 160
17.5 Admission Is a Different Judgment . . . 160

18 Containers and the Edge of the Process **163**
18.1 What a Container Is . . . 163
18.2 A Static Binary . . . 164
18.3 Running It . . . 165
18.4 What a Container Does Not Decide . . . 167
18.5 Where Part V Ends . . . 167

VI Experimental Systems **169**

19 Measurement **171**
19.1 Performance Claims Are Claims . . . 171
19.2 The Cost of an Append . . . 171
19.3 Where the Time Goes . . . 175
19.4 The Flush to Stable Storage . . . 175
19.5 The Index and the Pool . . . 176
19.6 What a Benchmark Cannot Say . . . 176

20 Logs and Faults **179**
20.1 Two Records With Different Jobs . . . 179
20.2 Faults as Inputs . . . 180
20.3 Seams and Simulated Faults . . . 181
20.4 An Error Is Not a Statement That Nothing Happened . . . 185

21 Manifests and Reproducibility **187**
21.1 What a Result Depends On . . . 187
21.2 Build Information . . . 192
21.3 Comparing Two Builds . . . 194
21.4 What the Manifest Does Not Establish . . . 194

22 What a Summary Discards **197**
22.1 The Question . . . 197
22.2 Summaries, Decisions, and Availability . . . 197
22.3 The Search . . . 198
22.4 Reading the Table . . . 207
22.5 The Digest That Answers Nothing . . . 208
22.6 What the Experiment Does Not Show . . . 208

Conclusion **211**

A Complete Source of the Command **213**

B Building and Running **229**

C An Inventory of Guarantees **231**

Preface

This book teaches the Go programming language by building one system from the first line to the last, instead of presenting a sequence of unrelated exercises. The system is a laboratory tool called `golab`. In its first form it is a command-line program that finds files in a research directory, reads them and records their cryptographic digests, runs external programs on them, and writes each of these actions into a ledger whose entries are chained together so that later alteration can be detected. Later parts of the book give the same tool a concurrent worker pool, durable indexes, an HTTP interface, container packaging, and a set of experimental harnesses. At every stage the program that exists is the program the chapter describes, and it compiles, passes its tests, and runs.

The conventional route into Go begins with variables and loops and arrives at web services by way of a long tour of the standard library. The route taken here is different because the intended reader is someone who already works fluently at a shell, who has written Bash scripts that traverse directories and invoke converters, and who has felt the point at which such scripts stop being adequate. The point is rarely a matter of speed. It arrives when a workflow must keep distinctions that the shell leaves implicit: which files were merely found and which were actually read, which outputs are evidence and which are interpretations of evidence, which failures left the filesystem partly changed, and which actions were permitted rather than merely performed. Go is a good language in which to write such distinctions down, because its types are simple enough to be read quickly, its errors are ordinary values that must be handled in plain sight, and its tools produce a single executable that can be run from the same shell the reader already uses.

Several ideas that organize the author’s wider research recur here as questions that shape the design of `golab`: the difference between a proposed transition and an accepted one, the difference between recording an event and authorizing it, the difference between a source and a summary of that source, and the difference between routing work to a destination and adjudicating whether the work should be done. These ideas are used as structural analogies and as prompts for design. The language does not enforce any of them automatically, and the book takes care to say, at each point, whether a given property is guaranteed by the compiler, guaranteed by a library under stated assumptions, or merely maintained by convention and discipline. A reader who finds that a chapter makes a stronger claim than its code supports should treat that as a defect of the chapter.

The book is organized in six parts. Part I, Local Computation, runs through six chapters. It introduces programs as transition systems, values and types, filesystem traversal and hashing, errors as values, the execution of external programs, and the hash-chained ledger. Part II, Governed Workflows, runs through three further chapters and introduces interfaces

and composition, together with a five-stage record pipeline in which storing, validating, admitting, projecting, and activating a record are separate steps with separate types. Part III, Concurrent Execution, runs through three more chapters and gives the tool a bounded worker pool, cancellation through contexts, and a ledger writer that serializes appends. Part IV, Memory and Reconstruction, runs through three more chapters and adds a file that is replaced atomically and an index that is derived from the ledger and can always be rebuilt. Part V, Distributed Systems, runs through three more chapters and gives the tool an HTTP interface, separates routing from adjudication and from admission, and runs the service in a minimal container. Part VI, Experimental Systems, runs through four chapters and treats the tool as an object of study, with benchmarks and profiles, structured logs and injected faults, manifests that fix what an experiment ran on, and a search for what a summary of a ledger discards. All six parts are complete in this volume.

Every listing in Parts I to VI is included directly from the source files of the `golab` module, so the text and the code cannot drift apart. The module was built and tested with Go 1.24 on Linux, and nothing in the whole book depends on a feature newer than Go 1.21. Readers working under the Windows Subsystem for Linux can follow the book without modification. Commands that appear in a shell listing were run, and the output shown is the output that was produced. Appendix C collects, in one table, which properties the compiler or toolchain guarantees and which are conventions, so that no claim in the text need be read as stronger than that table allows.

Part I

Local Computation

Chapter 1

A Program Is a Controlled Transition

1.1 The Case for a Compiled Tool

Go occupies a particular position among programming languages. It compiles to native executables, it manages memory automatically, it has structural interfaces, explicit error handling, and lightweight concurrency, and it ships with a standard library broad enough that many useful programs need no outside dependencies at all. Its syntax is deliberately restrained, which means that a reader can learn most of the language in a short time, but its operational reach is large. A program written in Go can traverse a filesystem, launch and supervise other programs, transform data, serve network requests, and run the same way on a laptop, in a scheduled job, and inside a container.

For a reader accustomed to Bash, Go at first looks unnecessarily elaborate. A shell script can list files, filter text, and invoke a converter in a handful of lines, whereas the equivalent Go program needs declarations, a compilation step, and explicit handling of every error. The elaboration pays for itself when a workflow has to preserve distinctions that a shell leaves unstated. Consider a research directory that holds source documents, generated PDFs, transcripts, measurements, and intermediate files. A script can find the files and call a converter on each. Suppose, however, that one conversion fails, that an output is older than its source, or that a machine-generated summary has been mistaken for the document it summarizes. The difficulty is no longer that of executing commands. It is the difficulty of maintaining a relation between operations, evidence, and the actions that the evidence permits afterward, and a tool that is expected to maintain such a relation needs a way to name its states, to refuse transitions that should not occur, and to leave a record of what it did. The shell remains the better instrument for brief interactive composition of commands, and nothing in this book proposes that it be abandoned. The proposal is narrower: when a sequence of convenient commands has become a system whose intermediate states and failure modes matter, it is time to write the system down in a language that can hold those states in its types.

1.2 The Smallest Program

The first program prints one line. It is reproduced here exactly as it exists in the `golab` module, under `examples/ch01_hello`.

```go
package main

import "fmt"

func main() {
    fmt.Println("Continuation begins.")
}
```

Listing 1.1: The minimal program.

A Go source file begins by naming its package. A package named `main` that defines a function named `main` is a program, and the function is where execution begins. The `import` declaration makes the standard package `fmt` available, and `fmt.Println` writes its arguments to standard output followed by a newline. Executed with `go run`, the program compiles and runs in one step.

```
$ go run ./examples/ch01_hello
Continuation begins.
```

Listing 1.2: Running the minimal program.

Unlike a Bash script, which is read and interpreted by a shell each time it runs, a Go program is compiled to machine code before it runs. The `go build` command performs the compilation alone and leaves an executable behind, which can then be run from a shell, scheduled by another tool, or copied into a container image. For programs that depend only on the standard library, that executable is ordinarily self-contained, although its portability is still bounded by the operating system and processor architecture for which it was built. A module, which is the unit in which Go code is versioned and built, is created with `go mod init`; the laboratory tool of this book is the module named `golab`, and every path in the listings is relative to its root.

1.3 Computation as a Transition System

A program can be described abstractly as a transition. Let S be the set of states the program can occupy, I the set of inputs it can receive, O the set of outputs it can produce, and E the set of exceptional outcomes it can report. A single step of the program is then a map

$$T : S \times I \longrightarrow S \times O \times E,$$

which takes a state and an input and yields a successor state together with an output and a possible error. The description is not peculiar to Go, and it is deliberately coarse, since it says nothing about how the step is carried out. Its use is that it separates three things that

a casual description of a program runs together: what the program remembers between steps, what it is asked to do, and what happens as a consequence.

The simplest nontrivial case is a counter. Its state is a single integer, and the operation it supports adds one to it. The version below is the example from the module under `examples/ch01_counter`.

```go
package main

import "fmt"

type State struct {
    Counter int
}

func Increment(s State) State {
    s.Counter++
    return s
}

func main() {
    current := State{Counter: 0}
    next := Increment(current)

    fmt.Println(current.Counter)
    fmt.Println(next.Counter)
}
```

Listing 1.3: A counter whose transition returns a new state.

The program prints `0` and then `1`. The point of the example is the manner in which `Increment` treats its argument. In Go, an argument of struct type is passed by value, so the function receives a private copy of `State`, increments the copy, and returns it. The variable `current` in `main` is not changed. The step therefore has the shape $s_{t+1} = T(s_t)$ with the old state still available for comparison, which is exactly the shape of the abstract definition. A program written in this style makes each transition visible as a function call whose input and output can both be inspected, and that visibility is what later chapters rely on when they attach a policy to a transition or a record to its outcome.

Remark. Process-oriented accounts of computation, including the SpherePOP calculus that appears elsewhere in the author's work, ask of any system not only which state it is in but which transitions it permits, which consequences of a transition persist, and which earlier histories remain reachable. That vocabulary suggests the questions this book asks of `golab`. It does not follow that Go implements any such calculus. Go supplies functions, values, and types, and the distinctions of a richer theory have to be encoded by the programmer in those materials.

1.4 Admissibility

Not every proposed transition should be accepted. Suppose the counter must never become negative. The requirement can be stated as an admissibility condition, a function A that inspects the current state and the proposed input and answers whether the transition may proceed:

$$A(s,i) = \begin{cases} 1 & \text{if the proposed transition is permitted,} \\ 0 & \text{otherwise.} \end{cases}$$

The transition is then defined only where $A(s,i) = 1$. Where $A(s,i) = 0$ the system reports a refusal and the state is left exactly as it was.

Go expresses refusal through ordinary return values. A function that can fail returns its result together with a value of the built-in type `error`, and the value `nil` means that no error occurred. The example below, from `examples/ch01_decrement`, applies the convention to the counter.

```go
package main

import (
    "errors"
    "fmt"
)

type State struct {
    Counter int
}

func Decrement(s State) (State, error) {
    if s.Counter == 0 {
        return s, errors.New("negative state prohibited")
    }

    s.Counter--
    return s, nil
}

func main() {
    current := State{Counter: 0}

    next, err := Decrement(current)
    if err != nil {
        fmt.Println("Rejected:", err)
        return
    }

    fmt.Println(next.Counter)
}
```

Listing 1.4: A decrement that refuses to produce a negative state.

Run with a counter of zero, the program prints `Rejected: negative state prohibited`. Two features of the example deserve attention. The first is that the distinction between a proposed transition and an accepted one is visible in the control flow: the call to `Decrement` proposes, the test of `err` decides, and only the branch in which `err` is `nil` proceeds to use the new state. The second is that on refusal the function returns the original state unchanged, so a caller who inspects the returned state after a failure sees the state from before the attempt. Both features are conventions of Go programming and neither is enforced by the compiler. A caller is free to write `next, _ := Decrement(current)` and continue as though nothing had been refused, and the compiler will accept the program. Static analysis tools outside the compiler can flag discarded errors, but the language itself does not require that an error be examined. This gap between reporting a failed transition and enforcing an admissibility boundary is the subject of the fourth exercise at the end of this chapter, and it returns in Chapter 4.

A more elaborate system will distinguish validation, authorization, and commitment, which are three different questions that are easily collapsed into one function. Validation asks whether the proposed input is well formed. Authorization asks whether the actor is entitled to request it. Commitment asks whether the resulting change is made durable. Part II returns to these separations. For now it is enough to note that the single function `Decrement` answers only the first of them and that nothing in the signature says so.

1.5 Admissibility Without Overflow

A bounded counter exposes a hazard that every program with arithmetic must face. Fix a maximum value M and require that the counter stay in the interval $[0, M]$. Adding a caller-supplied amount δ is then admissible when the sum lies in that interval. The obvious test computes $c + \delta$ and compares the result with the bounds, but in Go the signed integer types wrap around on overflow without any error or panic, so a sum that exceeds the largest representable integer silently becomes a large negative number, and a test that was meant to protect the bound can be defeated by the very arithmetic it performs. The test must therefore be arranged so that it never computes a quantity that can overflow.

```
// Package transition models a program as an explicit state transition
// system with an admissibility condition, as developed in Chapter 1.
package transition

import (
    "errors"
    "fmt"
)

// Max is the largest counter value the system admits.
const Max = 1000

// ErrNegative reports a proposed transition that would leave the
// counter below zero.
```

```
var ErrNegative = errors.New("negative state prohibited")

// ErrOverflow reports a proposed transition that would exceed Max.
var ErrOverflow = errors.New("state exceeds maximum")

// State is the whole state of the counter system.
type State struct {
    Counter int
}

// Increment returns a new state. The argument is copied on entry, so the
// caller's value is never modified.
func Increment(s State) State {
    s.Counter++
    return s
}

// Decrement proposes a transition and reports whether it is admissible.
func Decrement(s State) (State, error) {
    if s.Counter == 0 {
        return s, ErrNegative
    }
    s.Counter--
    return s, nil
}

// Add proposes adding delta to the counter. The admissibility test is
// written so that it cannot itself overflow: it compares delta against the
// remaining headroom instead of computing Counter+delta first. The lower
// bound is tested as delta < -Counter rather than -delta > Counter, because
// Counter is never negative, so -Counter is always representable, whereas
// -delta is not when delta is the most negative integer.
func Add(s State, delta int) (State, error) {
    switch {
    case delta >= 0 && delta > Max-s.Counter:
        return s, fmt.Errorf("add %d to %d: %w", delta, s.Counter, ErrOverflow)
    case delta < 0 && delta < -s.Counter:
        return s, fmt.Errorf("add %d to %d: %w", delta, s.Counter, ErrNegative)
    }
    s.Counter += delta
    return s, nil
}
```

Listing 1.5: The transition package, with an overflow-safe admissibility test.

The function `Add` in Listing 1.5 applies the principle on both sides of the interval. When $\delta \geq 0$ the transition is admissible exactly when $\delta \leq M - c$, and the right-hand side lies between 0 and M for every legitimate state c, so it cannot overflow. When $\delta < 0$ the transition is admissible exactly when $\delta \geq -c$, and since $c \geq 0$ the quantity $-c$ is always representable. The following proposition records the argument.

Proposition 1.1. *Let $0 \leq c \leq M$ and let δ be any integer in the representable range. Then $0 \leq c + \delta \leq M$ if and only if either $\delta \geq 0$ and $\delta \leq M - c$, or $\delta < 0$ and $\delta \geq -c$. Moreover, evaluating $M - c$ and $-c$ cannot overflow.*

Proof. If $\delta \geq 0$ then $c + \delta \geq 0$ automatically, and $c + \delta \leq M$ holds exactly when $\delta \leq M - c$. If $\delta < 0$ then $c + \delta \leq M$ automatically, and $c + \delta \geq 0$ holds exactly when $\delta \geq -c$. Because $0 \leq c \leq M$, the value $M - c$ lies in $[0, M]$ and the value $-c$ lies in $[-M, 0]$, and both intervals are contained in the representable range whenever M is. □

The reason for writing the lower-bound test as $\delta < -c$ and not as $-\delta > c$ is not cosmetic. In two's complement arithmetic the most negative integer has no positive counterpart, so negating it yields itself, and the apparently equivalent test accepts exactly the input that it should reject most firmly. The first version of `Add` written for this chapter used the negated form, and it was the table-driven test in Listing 1.6, which includes the most negative integer as a case, that rejected it. The episode is retained in the text because it illustrates the role of tests in the method of the book: a test is the means by which a claim about admissibility is checked against the arithmetic of the machine on which it must hold.

```
package transition

import (
    "errors"
    "math"
    "testing"
)

func TestIncrementLeavesOriginalUnchanged(t *testing.T) {
    current := State{Counter: 0}
    next := Increment(current)
    if current.Counter != 0 || next.Counter != 1 {
        t.Fatalf("got current=%d next=%d", current.Counter, next.
    Counter)
    }
}

func TestDecrementRejectsNegative(t *testing.T) {
    _, err := Decrement(State{})
    if !errors.Is(err, ErrNegative) {
        t.Fatalf("want ErrNegative, got %v", err)
    }
```

```
}

func TestAdd(t *testing.T) {
    cases := []struct {
        name  string
        start int
        delta int
        want  int
        err   error
    }{
        {"plain", 10, 5, 15, nil},
        {"to the maximum", 990, 10, 1000, nil},
        {"past the maximum", 990, 11, 990, ErrOverflow},
        {"below zero", 3, -4, 3, ErrNegative},
        {"to zero", 3, -3, 0, nil},
        {"huge positive", 1, math.MaxInt, 1, ErrOverflow},
        {"huge negative", 1, math.MinInt, 1, ErrNegative},
    }
    for _, c := range cases {
        t.Run(c.name, func(t *testing.T) {
            got, err := Add(State{Counter: c.start}, c.delta)
            if !errors.Is(err, c.err) {
                t.Fatalf("err = %v, want %v", err, c.err)
            }
            if got.Counter != c.want {
                t.Fatalf("counter = %d, want %d", got.Counter, c.want
    )
            }
        })
    }
}
```

Listing 1.6: Table-driven tests for the transition package.

The test file shows the standard shape of a Go test. A function whose name begins with `Test` and which takes a pointer to `testing.T` is run by the `go test` command. The table in `TestAdd` lists cases as data, and each case is run as a named subtest, so that a failure identifies the case that failed. The comparison `errors.Is(err, c.err)` asks whether the returned error is, or wraps, the expected sentinel value, a technique taken up in Chapter 4.

1.6 From Transitions to Workflows

The step from a counter to a workflow is a change of subject matter and not of method. Suppose that the task is to list the regular files below a directory. In Bash the answer is a single command, `find . -type f`. The corresponding Go program, from `examples/ch01_walk`, is considerably longer.

```
package main
```

```
2
3  import (
4      "fmt"
5      "io/fs"
6      "path/filepath"
7  )
8
9  func main() {
10     err := filepath.WalkDir(".", func(path string, entry fs.DirEntry,
       walkErr error) error {
11         if walkErr != nil {
12             return walkErr
13         }
14         if !entry.IsDir() {
15             fmt.Println(path)
16         }
17         return nil
18     })
19     if err != nil {
20         fmt.Println("Traversal failed:", err)
21     }
22 }
```

Listing 1.7: Listing regular files below the current directory.

The program hands `filepath.WalkDir` a function to be called for every entry. The function receives the path, a description of the entry, and any error encountered in reaching it, and it returns an error to stop the traversal or `nil` to continue. The length of the program relative to the shell command is real, but what the extra length buys is a place to attach things the shell command has no room for: a typed description of each file, a classification, a content hash, a policy about unreadable directories, and a record of the traversal. Chapter 3 develops those additions, and by the end of Part I the same traversal logic sits behind commands such as `golab discover`, `golab verify`, and `golab run`.

A tool of this kind has four responsibilities which can coexist in one executable but should remain distinguishable in its design. A command is an instruction that the tool accepts from its operator. An operation is the work performed in response to a command. A policy is the rule that determines whether the work is permitted. A record is the durable trace that preserves what happened. In the version of `golab` developed in Part I the commands are implemented in the `cmd/golab` package, the operations in the packages `artifact` and `runner`, the records in the package `ledger`, and the policy, which is still rudimentary, in the classification rules and in the admissibility checks of the transition package. Part II gives policy a package of its own. The ordering of the book follows from this principle: distribution changes the circumstances in which computation takes place, but it does not remove the need to say what the computation is. Goroutines, channels, network services, and containers are therefore deferred until the local tool is understood, since each of them multiplies the consequences of whatever ambiguity the local tool still contains.

Exercises

Exercise 1.1. Modify the counter of Listing 1.3 so that the caller supplies an increment amount, and reject any transition that would exceed a fixed maximum. Write the admissibility test so that it cannot overflow, state the argument for its correctness in the manner of Proposition 1.1, and add table-driven tests that include the largest and smallest representable integers.

Exercise 1.2. Modify the traversal program of Listing 1.7 so that it prints only files whose names end in `.go`, and then add a count of the matching files that is reported after the traversal completes. Decide what the program should do when a subdirectory cannot be read, and justify the choice.

Exercise 1.3. Define two distinct Go types, one for a file that has been discovered and one for a file whose content has been verified, and consider which functions ought to accept each. Explain what, if anything, prevents a caller from manufacturing a value of the second type without performing any verification. Compare your answer with the treatment in Chapter 2.

Exercise 1.4. Explain why returning an error from a function does not by itself guarantee that the caller will respond correctly. State the difference between reporting a failed transition and enforcing an admissibility boundary, and describe one mechanism, inside or outside the language, that moves a program from the former toward the latter.

Exercise 1.5. Compare a Bash pipeline with a Go program that performs the same sequence of operations on a set of files. For each, identify which intermediate states are explicit, which errors can be observed by the operator, and which failures could leave the filesystem partly changed. Consider in particular what happens when the third of five steps fails.

Chapter 2

Values, Types, and Records

2.1 Structs, Copies, and Zero Values

A Go program describes its data with types, and the most important composite type for the purposes of this book is the struct, a fixed collection of named fields. The counter of the previous chapter used a struct with a single field, which may have seemed a needless ceremony, but the habit of wrapping state in a named type pays off as soon as the state grows. The artifact package of `golab` begins with a struct that describes a file that has been found. The struct records three things only, the path, a classification, and the size in bytes at the moment of the walk. It does not record a modification time, an owner, or any property of the content.

```
// Package artifact separates files that have merely been found from files
// whose content has been read and hashed. The separation is carried by the
// type system: code that requires evidence of content can ask for a
// Verified value instead of a path.
package artifact

import (
    "path/filepath"
    "strings"
)

// Kind classifies an artifact by role. A summary is deliberately a
// different kind from a source: it is an interpretation of evidence, not
// evidence.
type Kind string

const (
    KindSource  Kind = "source"
    KindOutput  Kind = "output"
    KindSummary Kind = "summary"
```

```
    KindOther   Kind = "other"
)

// Discovered is a path that exists. Nothing is claimed about its content.
type Discovered struct {
    Path string
    Kind Kind
    Size int64
}

// Verified is a Discovered file whose bytes were read and hashed. Its
// fields are unexported, so code outside this package cannot assemble one
// with an invented digest. Go always permits the zero value, however, so a
// Verified whose digest is empty must be treated as "nothing verified".
type Verified struct {
    found Discovered
    sum   string
}

// Path returns the location of the verified file.
func (v Verified) Path() string { return v.found.Path }

// Kind returns the classification of the verified file.
func (v Verified) Kind() Kind { return v.found.Kind }

// Size returns the number of bytes that were hashed.
func (v Verified) Size() int64 { return v.found.Size }

// SHA256 returns the hexadecimal digest, or the empty string for the zero
// value.
func (v Verified) SHA256() string { return v.sum }

// IsZero reports whether v is the zero value and therefore carries no
// evidence of any content.
func (v Verified) IsZero() bool { return v.sum == "" }

// Classify assigns a Kind from the file name alone. It is a convention,
// and is documented as one in Chapter 3.
func Classify(path string) Kind {
    base := strings.ToLower(filepath.Base(path))
    switch {
    case strings.HasSuffix(base, ".summary.md") || strings.HasSuffix(base, ".summary.txt"):
```

```
63          return KindSummary
64      case strings.HasSuffix(base, ".tex") || strings.HasSuffix(base, ".
        go") ||
65          strings.HasSuffix(base, ".md") || strings.HasSuffix(base, ".
        txt"):
66          return KindSource
67      case strings.HasSuffix(base, ".pdf") || strings.HasSuffix(base, ".
        html"):
68          return KindOutput
69      default:
70          return KindOther
71      }
72  }
```

Listing 2.1: The types of the artifact package.

Three properties of structs govern almost every design decision in the chapters that follow. First, structs are values: assigning one to another variable, passing it to a function, or returning it from a function copies its fields, so the receiver holds an independent value. Second, every type has a zero value that is available without any explicit construction, and for a struct the zero value is the struct whose every field has its own zero value, which is the empty string for a string, zero for a number, and `nil` for a pointer, slice, or map. Third, a composite literal such as `Discovered{Path: p, Kind: k, Size: n}` names the fields it sets and leaves the rest at their zero values, so that adding a field to a struct does not break existing literals that name their fields.

The zero value is a convenience in many programs and a hazard in programs that wish to say that a value has not been established. A `Discovered` with an empty path is not a meaningful description of any file, but the language will construct one whenever asked, and it is the responsibility of the surrounding code to refuse to treat it as a discovery. This tension between what the type system can express and what the design requires is one of the main themes of Part I, and Section 2.5 returns to it.

2.2 Named Types as Vocabularies

The type `Kind` is defined as a string with a name of its own, and four constants give it its intended vocabulary: source, output, summary, and other. Naming the type has two benefits. A function that takes a `Kind` says in its signature what sort of string it expects, and the compiler refuses to pass a plain `string` variable where a `Kind` is wanted without an explicit conversion, which catches a class of mistakes in which one kind of text is confused with another.

The benefit has a limit that the book states plainly, because the limit matters to the design. Go has no enumeration type that closes a vocabulary. A `Kind` is a string underneath, the expression `Kind("anything")` is a legal conversion, and the compiler will not object to a value outside the four declared constants. The closure of the vocabulary is therefore a convention, maintained by declaring the constants in one place, by routing classification through one function, and by having any consumer that switches on the kind treat an unrecognized value as an error and not as a default. The convention is cheap and effective

in a small program, and it is worth being clear that it is a convention. A guarantee would require either a type whose representation cannot be forged from outside its package, which is the technique used for `Verified` below, or a runtime check at each boundary.

The vocabulary contains a deliberate distinction between a source and a summary. A file ending in `.summary.md` is classified as a summary, and a file ending in `.tex` as a source. The classification is made from the name alone and is therefore weak evidence, as Chapter 3 discusses, but the existence of the distinction in the type is what allows later operations to decline to treat a generated summary as if it were the document it summarizes. A summary is an interpretation of evidence and not itself evidence, and a system that can say so in its vocabulary can enforce it in its policies.

2.3 Methods and the Package Boundary

A method is a function with a receiver, written between the keyword `func` and the method name. The methods on `Verified` in Listing 2.1 are accessors, and they exist because of the way Go controls visibility. A name that begins with an upper-case letter is exported and may be used by other packages, while a name that begins with a lower-case letter is private to its own package. The fields of `Verified` are named `found` and `sum`, in lower case, so no code outside the artifact package can set them or read them directly. Code elsewhere can obtain a `Verified` only from the function `Verify`, which lives in the same package, or as the zero value, and can inspect it only through the accessor methods.

This is a genuine guarantee, and it differs in kind from the conventions discussed above. The Go compiler enforces the visibility rule, so a package that imports `artifact` cannot write `artifact.Verified{sum: "abc"}` and have it compile. The guarantee is limited in two ways that should be understood exactly. It does not prevent the zero value, which any code may obtain by declaring `var v artifact.Verified`, and which is why the type carries an `IsZero` method and why the documentation says that a zero value carries no evidence. The method is specific to `Verified`, the one type in the module whose zero value would be mistaken for a successful act. The other types do not need it, because their zero values are either harmless, as Chapter 8 shows for `Valid`, or are already rejected by an existing check. It also says nothing about whether `Verify` itself is correct, since the type records only that a digest was produced by that function and cannot check that the digest corresponds to the bytes on disk at the moment of use. A file can change after it was verified. What the type can claim is that someone, through the one sanctioned route, read the bytes once and recorded what they hashed to.

2.4 Packages, Modules, and the Internal Directory

The `golab` module is organized so that each concern lives in a package of its own. The command-line entry point is in `cmd/golab`, and the reusable machinery lives under `internal`. The directory name `internal` has a meaning that the `go` tool itself enforces: a package under `internal` can be imported only by code rooted at the parent of that directory. In this module, therefore, `golab/internal/artifact` can be imported by the command and by other packages of `golab` but not by any outside module, and the author

of the tool is free to change those packages without breaking users who never had access to them. As with the visibility of fields, this is a mechanical guarantee supplied by the toolchain and not a convention.

Dividing the code this way supports the separation of roles introduced in Chapter 1. The package that finds and hashes files knows nothing of the command line, and the package that writes the ledger knows nothing of files. A package that does not import another cannot depend on it, and the Go compiler rejects import cycles, so the dependency structure of the program is a directed acyclic graph that the toolchain checks on every build. The graph for Part I is simple: the command imports three packages, and none of the three imports another.

2.5 Two Types for Two States of Knowledge

The pair `Discovered` and `Verified` encodes a distinction that the shell cannot state. A path returned by `find` is a claim about the existence of a directory entry at one moment and carries no commitment about what the entry contains. A digest of the bytes of a file, computed by reading them, is a claim of a different strength: it supports the later assertion that a file with that digest has exactly those contents, to within the collision resistance of the hash function. By giving the two states different types, the design makes it possible for a function to declare which it needs. An operation that merely lists files can take a `Discovered`, and an operation that is going to treat the content of a file as established, by recording it in a ledger or comparing it with an earlier digest, can require a `Verified`, so that a call that skips the reading step does not compile.

The design should not be oversold. It does not make the program correct, and it does not stop a careless author from verifying a file and then ignoring the digest. What it does is move one class of mistake, using an unverified file where a verified one is required, from the category of runtime surprises to the category of compile-time errors, and it records in the program text a commitment about what each stage has and has not established. The same technique recurs throughout the book, and in Part II it is applied to records and to the stages of the ledger pipeline.

Exercises

Exercise 2.1. Add a method `Valid` to the type `Kind` that reports whether a value is one of the four declared constants. Decide where in `golab` the method should be called, and write a test that fails if a new constant is added to the vocabulary without being added to `Valid`.

Exercise 2.2. The zero value of `Verified` carries no evidence. Write a function that accepts a `Verified` and returns an error when given the zero value, and explain whether this check belongs in the function, in its callers, or in the type. Consider the cost of forgetting the check in each placement.

Exercise 2.3. Replace the exported fields of `Discovered` with unexported fields and accessors, and observe which parts of the program must change. Argue for or against this

change, taking into account that a `Discovered` makes no claim about the content of the file.

Exercise 2.4. Draw the import graph of the `golab` module as it stands at the end of Part I. Explain why the Go compiler's refusal of import cycles constrains the placement of the type that represents a ledger record relative to the code that constructs it.

Chapter 3

Traversing, Reading, and Hashing

3.1 Walking a Directory Tree

The operations of the artifact package are in a file of their own, shown below. The first of them, `Discover`, is the typed descendant of the `find` command from Chapter 1.

```
package artifact

import (
    "crypto/sha256"
    "encoding/hex"
    "fmt"
    "io"
    "io/fs"
    "os"
    "path/filepath"
    "sort"
    "strings"
)

// Discover walks root and returns every regular file in lexical order.
// Directories whose names begin with a dot are skipped.
func Discover(root string) ([]Discovered, error) {
    var found []Discovered
    err := filepath.WalkDir(root, func(path string, entry fs.DirEntry, walkErr error) error {
        if walkErr != nil {
            return walkErr
        }
        if entry.IsDir() {
            if path != root && strings.HasPrefix(entry.Name(), ".") {
                return filepath.SkipDir
            }
            return nil
        }
```

```
        if !entry.Type().IsRegular() {
            return nil
        }
        info, err := entry.Info()
        if err != nil {
            return err
        }
        found = append(found, Discovered{Path: path, Kind: Classify(path), Size: info.Size()})
        return nil
    })
    if err != nil {
        return nil, fmt.Errorf("discover %s: %w", root, err)
    }
    sort.Slice(found, func(i, j int) bool { return found[i].Path < found[j].Path })
    return found, nil
}

// Verify reads the file and records its SHA-256 digest.
func Verify(d Discovered) (Verified, error) {
    f, err := os.Open(d.Path)
    if err != nil {
        return Verified{}, fmt.Errorf("verify %s: %w", d.Path, err)
    }
    defer f.Close()

    h := sha256.New()
    n, err := io.Copy(h, f)
    if err != nil {
        return Verified{}, fmt.Errorf("verify %s: %w", d.Path, err)
    }
    if n != d.Size {
        return Verified{}, fmt.Errorf("verify %s: size changed from %d to %d during read", d.Path, d.Size, n)
    }
    return Verified{found: d, sum: hex.EncodeToString(h.Sum(nil))}, nil
}
```

Listing 3.1: Discovering and verifying files.

The function `filepath.WalkDir` visits every entry beneath a root, calling the supplied function once for each. The function can steer the traversal through its return value. Returning `nil` continues, returning the special value `filepath.SkipDir` when positioned on a directory prevents the traversal from descending into it, and returning any other error stops the whole walk and delivers that error to the caller. `Discover` uses all three behaviours. It skips directories whose names begin with a dot, so that version-control metadata and similar private directories are not mistaken for research material, but it exempts the root itself from the rule so that a caller who names a directory beginning with a

dot is not silently handed an empty result. It propagates any error that the walk reports, so that an unreadable subdirectory fails the discovery instead of shrinking it without notice. And it ignores entries that are not regular files, such as symbolic links, sockets, and devices, which is a policy choice stated in the code rather than left to accident.

The choice to fail on an unreadable subdirectory is a decision about policy, and it is worth seeing that it is only one of several defensible decisions. A tool that is used interactively to explore a large tree might prefer to skip what it cannot read and report the omissions at the end. A tool whose purpose is to establish a complete inventory of evidence cannot accept a silently partial inventory, since a later conclusion that something is absent from the directory would then rest on a traversal that never looked. `golab` takes the second view, and the first exercise at the end of the chapter asks for the alternative.

Two details of the implementation repay attention. The documentation of `WalkDir` states that it visits entries of each directory in lexical order, but the order of whole paths produced by a depth-first walk is not the same as the lexical order of the complete path strings, because a directory name that is a prefix of a sibling's name interleaves differently. The function therefore sorts the final slice by path, so that the result is deterministic and independent of the details of the walk. Deterministic output matters here because the ledger will record the results of operations, and an operation whose output order varies from run to run would produce records that cannot be compared.

3.2 Reading Without Loading

The function `Verify` reads a file and records its digest. The central line is the call to `io.Copy`, which copies from the open file into the hash. That the call is possible at all is a consequence of one of the most characteristic features of Go, the structural interface. The value `h`, returned by `sha256.New`, implements the interface `hash.Hash`, which includes the method `Write`, and therefore it can be used wherever an `io.Writer` is expected, with no declaration anywhere that says so. A type satisfies an interface in Go merely by having the required methods. The consequence for this program is that the file is streamed through the hash in blocks, and the program never holds the whole file in memory, so that verifying a ten-gigabyte measurement file costs the same memory as verifying a ten-line note.

The deferred call `defer f.Close()` schedules the closing of the file to occur when the surrounding function returns, whichever way it returns. Without it, every early return would need its own call to `Close`, and a forgotten one would leak a file descriptor. A deferred call is run at function exit in the reverse order of its registration, and its arguments are evaluated at the moment of the `defer` statement. The idiom is so common that it is the first thing a reader of Go code looks for after a successful call to `os.Open`. For a file opened only for reading, ignoring the error that `Close` can return is conventional and harmless. For a file opened for writing the same shortcut can lose a failure that matters, and the ledger package, in Chapter 6, checks the error of `Close` explicitly for that reason.

The size comparison after the copy is a small defence against a particular failure. The `Discovered` value carries the size that the file had when the walk observed it, and `Verify` compares it with the number of bytes actually hashed. If the two differ, the file was modified between discovery and verification, and the function reports the discrepancy and returns

no digest. This check is genuinely useful and also genuinely limited. It detects a change in length, and it does not detect a change that preserves length. The gap between a check made at one moment and a use made at another is a general feature of programs that work on a shared filesystem and is known in the security literature as a time-of-check to time-of-use race. `golab` does not close it and does not claim to. The digest it records attests that the bytes read at a particular time hashed to a particular value, and nothing more.

3.3 What a Digest Establishes

The hash function SHA-256 maps a sequence of bytes of any length to a fixed-length digest of 256 bits, and it is designed so that finding two different inputs with the same digest is computationally infeasible. Under that assumption a recorded digest is strong evidence that a file later found to have the same digest has the same bytes, and equally strong evidence that a file with a different digest differs in at least one byte. That is the entire content of the claim. A digest does not say who wrote the file, whether its contents are true, whether the file is the one the author intended, or whether it is the original from which a copy was made. It is evidence about identity of content and about nothing else.

The point is stated carefully because of a temptation that arises in any tool that records digests. Having computed and stored a digest, it is easy to speak as though the tool had verified the document, and the word invites the further inference that the document is therefore reliable. In `golab` the type `Verified` is named for a narrow fact, that the bytes were read and hashed, and the narrow name is deliberate. Any stronger claim would need further evidence, such as a signature by a known key or a comparison with a digest published by an independent party, and the ledger in Chapter 6 is designed so that such evidence can be added without changing the meaning of what is already recorded.

The same reasoning explains why `Classify` inspects only file names. A name ending in `.summary.md` is a convention adopted by the operator, and a file with that name might, through error or intent, contain the original text of a source. The classification is documented in the code as a convention, and it is useful because it is cheap and usually right. It is not a verification of the role of the file, and a stricter tool would record, alongside the classification, how it was obtained.

3.4 Testing Against the Filesystem

The tests for the artifact package build their own small directories, and doing so introduces a technique worth learning early. The method `t.TempDir` returns a fresh directory that the testing framework removes when the test finishes, so a test can create whatever files it needs without risk to the working tree and without cleaning up. The helper function `write` in the test file creates intermediate directories and writes a file, and calls `t.Helper` so that a failure inside it is reported at the line of the test that called it. The first test builds a directory containing one source, one output, one summary, and a hidden directory, and checks both that the hidden file is skipped and that each of the others receives the intended classification.

```
package artifact

import (
    "errors"
    "io/fs"
    "os"
    "path/filepath"
    "testing"
)

func write(t *testing.T, dir, name, body string) {
    t.Helper()
    p := filepath.Join(dir, name)
    if err := os.MkdirAll(filepath.Dir(p), 0o755); err != nil {
        t.Fatal(err)
    }
    if err := os.WriteFile(p, []byte(body), 0o644); err != nil {
        t.Fatal(err)
    }
}

func TestDiscoverClassifiesAndSkipsHidden(t *testing.T) {
    dir := t.TempDir()
    write(t, dir, "paper.tex", "x")
    write(t, dir, "paper.pdf", "x")
    write(t, dir, "paper.summary.md", "x")
    write(t, dir, ".git/config", "x")

    got, err := Discover(dir)
    if err != nil {
        t.Fatal(err)
    }
    if len(got) != 3 {
        t.Fatalf("found %d files, want 3: %+v", len(got), got)
    }
    want := map[string]Kind{
        "paper.pdf":         KindOutput,
        "paper.summary.md": KindSummary,
        "paper.tex":         KindSource,
    }
    for _, d := range got {
        if want[filepath.Base(d.Path)] != d.Kind {
            t.Errorf("%s classified %s", d.Path, d.Kind)
        }
    }
}

func TestVerifyKnownDigest(t *testing.T) {
    dir := t.TempDir()
    write(t, dir, "a.txt", "abc")
    found, err := Discover(dir)
```

```
    if err != nil {
        t.Fatal(err)
    }
    v, err := Verify(found[0])
    if err != nil {
        t.Fatal(err)
    }
    const abc = "ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad"
    if v.SHA256() != abc || v.IsZero() {
        t.Fatalf("digest %s", v.SHA256())
    }
}

func TestZeroVerifiedCarriesNoEvidence(t *testing.T) {
    var v Verified
    if !v.IsZero() || v.SHA256() != "" {
        t.Fatalf("zero value should carry no digest: %+v", v)
    }
}

func TestVerifyMissingFileKeepsItsCause(t *testing.T) {
    _, err := Verify(Discovered{Path: filepath.Join(t.TempDir(), "gone.txt")})
    if err == nil {
        t.Fatal("expected an error for a missing file")
    }
    if !errors.Is(err, fs.ErrNotExist) {
        t.Fatalf("the cause was lost in wrapping: %v", err)
    }
    var pe *fs.PathError
    if !errors.As(err, &pe) || pe.Op != "open" {
        t.Fatalf("want a *fs.PathError for the open call, got %v", err)
    }
}
```

Listing 3.2: Tests for discovery and verification.

The second test checks the digest of the three bytes `abc` against the published test vector for SHA-256, which begins `ba7816bf` and is a standard value in the specification of the algorithm. Testing against a known external value is stronger than testing that the function returns the same thing twice, because it checks the function against an independent source of truth. The third test confirms that a missing file produces an error. The remaining test, added in Chapter 2, checks that the zero value of `Verified` claims nothing.

Exercises

Exercise 3.1. Modify `Discover` so that a directory that cannot be read is recorded in a returned list of omissions and the traversal continues. Change the signature of the function accordingly, and write a test that makes a directory unreadable. Discuss which behaviour a tool used to certify the absence of a file ought to have.

Exercise 3.2. Symbolic links are currently ignored. Extend the discovery to report them as a separate kind, without following them, and explain what could go wrong if the walk followed links in a tree that contains a cycle.

Exercise 3.3. Construct a scenario in which a file is modified between discovery and verification and the modification is not detected by the size check. Describe a stronger check that would catch it, and state what it would cost.

Exercise 3.4. Replace SHA-256 with a different hash function from the standard library, changing as little of the program as possible. Explain which parts of the program had to change, and what that reveals about the role of interfaces in the design.

Chapter 4

Errors as Values

4.1 The Error Interface

In many languages a failure interrupts the normal flow of control and travels up the call stack until some enclosing handler catches it. Go makes a different bet. A failure is an ordinary value of the built-in interface type `error`, whose entire definition is a single method that returns a string, and a function that can fail returns such a value alongside its result. The caller receives the value at the point of the call and must decide, in plain sight, what to do with it. The cost of the arrangement is verbosity, since the test `if err != nil` appears with a frequency that new readers find tiresome. The benefit is that every place where a failure can occur is visible in the text of the program, and every place where one is ignored is visible as well.

The previous chapters used two constructors for errors. The function `errors.New` produces an error with a fixed message, and the function `fmt.Errorf` produces one whose message is formatted from its arguments. A third feature of `fmt.Errorf` is the one that matters most for the design of `golab`: the verb `%w` wraps another error, so that the new error carries both the added context and the original. In the artifact package, a failure to open a file is reported as an error whose text is of the form `verify path: open path: no such file or directory`. The outer message says what `golab` was attempting and the inner one says what the operating system refused, and both are present in the one value.

4.2 Sentinel Errors and Inspection

Wrapping would be of little use if the only way to examine a wrapped error were to search its text. The standard library provides two functions for inspecting the chain. The function `errors.Is` asks whether any error in the chain is equal to a given target value, and it is the usual way to test for a sentinel, which is a package-level error variable that names a condition. The transition package declares two of them, `ErrNegative` and `ErrOverflow`, and the ledger package declares `ErrBroken`. A caller that wishes to treat an overflow differently from a negative result can write `errors.Is(err, transition.ErrOverflow)`, and the test succeeds even though the error that `Add` returns has been wrapped with the

numbers involved.

The second function, `errors.As`, asks whether any error in the chain has a given type and, if so, assigns it to a variable of that type so that its fields can be read. The test `TestVerifyMissingFileKeepsItsCause`, in Listing 3.2, uses both. It verifies a file that does not exist and then checks that `errors.Is(err, fs.ErrNotExist)` holds, which shows that the underlying cause survived the wrapping performed by `Verify`, and that `errors.As` can recover the `*fs.PathError` produced by the operating-system layer, whose `Op` field is `"open"`. The point of the test is to protect a property that would otherwise erode unnoticed. A later change that formatted an error with `%v` in place of `%w` would leave every message looking the same to a human reader while severing the chain, and the callers that depend on `errors.Is` would begin failing to recognize conditions they had previously recognized.

A sentinel is part of the public interface of the package that declares it, in the same way that a function is, and it should be introduced with the same care. Once callers test for `ErrBroken`, the package has committed to returning it, wrapped or not, in the circumstances that the documentation describes. The sentinels in `golab` are few and each names a distinct refusal: a transition that would leave the permitted interval on the low side, one that would leave it on the high side, and a ledger whose chain of digests does not hold.

4.3 Context and the Cost of Silence

Error values are most useful when they carry enough context to be acted upon and least useful when they say only that something failed. The functions of `golab` follow a convention that each layer prefixes the error it returns with the operation it was performing and the object of the operation, so that by the time an error reaches the command it reads as a short narrative. The text `golab: record 1 digest mismatch: ledger chain broken` shows the pattern. The ledger names the record, the failed check, and the sentinel, and the command adds its own name. A convention of this kind has the property that no layer needs to know the details of another. The artifact package does not know that its errors will be printed by a command, and the command does not know the structure of the errors it prints.

The opposite habit, returning the inner error unmodified, loses the narrative, and the habit of discarding it altogether is worse. The blank identifier _ lets a caller accept a result and ignore the error that accompanied it, and the compiler permits the practice. Static analysers can warn about it, and a project that cares about its failure modes will enable them, but the language places no obstacle in the way. The consequence is that Go provides a convention for handling errors and not a mechanism that forces it. The distinction between reporting a failure and enforcing the boundary the failure was supposed to protect, raised in the fourth exercise of Chapter 1, has its practical form here: a function that returns `ErrNegative` reports that a transition was refused, and the refusal has an effect on the system only if the caller refrains from using the state it would have produced.

4.4 Partial Failure as Data

Operations that act on many files raise a question that single operations do not, namely what to do when some succeed and some fail. The `verify` command of `golab` makes a choice that is worth examining, because it is the choice of a tool that is meant to keep records. It attempts every file, collects the failures instead of stopping at the first, and then appends one record to the ledger that states how many files were found, how many were verified, and how many failed. The record is an aggregate: it holds three counts and does not list the individual files or their outcomes. Per-file results are printed to the terminal and are not stored in the ledger, a limitation that Part III removes with an optional flag. Only after the record has been written does it return an error if any file failed. The order matters. A command that returned at the first failure would leave the ledger without any record that the attempt had been made, and a command that reported success after any failure would make the record misleading. The chosen order ensures that the ledger always contains an account of what was tried and what happened, and that the exit status of the program tells the operator that the account contains failures.

The behaviour is a small instance of a general principle for systems that must be audited. Failure is information, and a system that records only its successes gives an auditor a distorted picture of what it does. The commands of `golab` are therefore built so that refusals, failures, and timeouts are written to the ledger in the same form as successes, and the program's exit status is a separate matter, reserved for communicating with the shell.

4.5 Errors and Panics

Go also has a mechanism for conditions that the program cannot reasonably continue from. The built-in function `panic` stops the ordinary flow of control, runs any deferred calls, and terminates the program with a stack trace unless a deferred function recovers. The runtime raises a panic for errors such as indexing past the end of a slice or dereferencing a nil pointer. The convention of the language is to reserve panics for programmer error and for impossible states, and to use returned errors for anything that can happen in a correctly written program running in an imperfect world, which includes missing files, malformed input, and refused permissions. No code in Part I calls `panic`, and the `main` function of `golab` does not recover from one. A panic in a tool of this kind is a bug to be fixed, and hiding it would obscure the very evidence that is needed to fix it.

Exercises

Exercise 4.1. Change the `%w` in the `Verify` function to `%v` and run the test suite. Identify which test fails and explain what property it was protecting.

Exercise 4.2. Define a typed error in the transition package that carries the attempted amount and the current counter, and arrange for `Add` to return it wrapping the appropriate sentinel. Show that a caller can recover both the sentinel and the numbers. Discuss when a sentinel is preferable to a typed error and when it is not.

Exercise 4.3. The `verify` command appends its record to the ledger before returning an error for failed files. Describe a failure of the ledger itself that would defeat this ordering, and propose a policy for the command to follow in that case.

Exercise 4.4. Install a static analyser that reports discarded errors, run it against the module, and report what it finds. Then deliberately introduce a discarded error and observe the report. Explain why the analyser is a better description of a policy than of a language feature.

Chapter 5

Running Programs Without a Shell

5.1 The Argument Vector

A workflow tool spends much of its life invoking other programs, and the manner of invocation determines how much of the shell’s fragility the tool inherits. When a Bash script runs a command, it hands a line of text to an interpreter that splits it into words, expands variables, performs globbing, and interprets quotation, and every one of those steps is an opportunity for a file name containing a space, a quotation mark, or a dollar sign to be read as something other than a name. Go’s `os/exec` package avoids the whole class. The function `exec.Command` takes the program and each argument as separate strings and passes them to the operating system as the argument vector, with no intervening interpretation. A file called `my paper (final).tex` is one argument, and it remains one argument however the program is run.

The package `runner` wraps the facility in a form suited to a tool that must record what happened.

```
// Package runner executes an external program and returns its outcome as
// a value, so that a nonzero exit status, a timeout, and a failure to start
// are three distinguishable results rather than one undifferentiated error.
package runner

import (
    "bytes"
    "context"
    "errors"
    "fmt"
    "os/exec"
    "time"
)

// Outcome describes how a run ended.
type Outcome string
```

```
17
18 const (
19     Succeeded Outcome = "succeeded"
20     Failed    Outcome = "failed"
21     TimedOut  Outcome = "timed-out"
22     NotStated Outcome = "did-not-start"
23 )
24
25 // Result is everything observed about one run.
26 type Result struct {
27     Outcome  Outcome
28     ExitCode int
29     Stdout   string
30     Stderr   string
31     Elapsed  time.Duration
32 }
33
34 // Run starts name with args in dir and waits for it, killing it when
       the
35 // timeout elapses. A non-nil error is returned only for conditions
      in which
36 // no meaningful Result exists; every other ending is reported in the
       Result.
37 func Run(dir string, timeout time.Duration, name string, args ...
      string) (Result, error) {
38     ctx, cancel := context.WithTimeout(context.Background(), timeout)
39     defer cancel()
40
41     cmd := exec.CommandContext(ctx, name, args...)
42     cmd.Dir = dir
43     // When the context expires only the started process is killed. A
44     // grandchild may still hold the output pipes open, so Wait is
      given a
45     // bounded grace period instead of waiting for the pipes
      indefinitely.
46     cmd.WaitDelay = 500 * time.Millisecond
47     var stdout, stderr bytes.Buffer
48     cmd.Stdout = &stdout
49     cmd.Stderr = &stderr
50
51     start := time.Now()
52     err := cmd.Run()
53     res := Result{
54         Stdout:  stdout.String(),
55         Stderr:  stderr.String(),
56         Elapsed: time.Since(start),
57     }
58
59     var exitErr *exec.ExitError
60     switch {
61     case err == nil:
```

```
        res.Outcome = Succeeded
    case errors.Is(ctx.Err(), context.DeadlineExceeded):
        res.Outcome = TimedOut
        res.ExitCode = -1
    case errors.As(err, &exitErr):
        res.Outcome = Failed
        res.ExitCode = exitErr.ExitCode()
    default:
        res.Outcome = NotStated
        res.ExitCode = -1
        return res, fmt.Errorf("run %s: %w", name, err)
    }
    return res, nil
}
```

Listing 5.1: Running an external program and classifying the outcome.

The function `Run` starts a program in a given directory, waits for it, and returns a `Result` that holds the outcome, the exit code, the captured output and error streams, and the elapsed time. Capturing is arranged by assigning `bytes.Buffer` values to the `Stdout` and `Stderr` fields of the command. Because those fields are of type `io.Writer`, the buffers satisfy them by the same structural rule that allowed a hash to be written to in the previous chapter, and the program is spared the intermediate temporary files that a shell script would use for the same purpose.

5.2 Four Endings

A shell script usually distinguishes two ways in which a command can end, by testing whether its status `$?` is zero. A tool that keeps records needs a finer classification, because the operator will ask different questions about different endings. A program that ran and exited with a nonzero status has told us something about its work. A program that was stopped by the tool because it ran too long has told us nothing, and the tool has made a decision on its behalf. A program that could not be started at all, because it does not exist or is not executable, has never run. The type `Outcome` names the four cases of success, failure, timeout, and failure to start, and the `switch` in `Run` sorts every possible ending into exactly one of them.

Go has no sum types, so the four cases are represented as string constants of a named type, with the closure of the set maintained by convention as discussed in Chapter 2. The `switch` makes the order of the tests significant. The timeout is tested before the exit-error case because a process killed by the expiry of the context also yields an error of the exit type, and testing the cause first prevents a killed process from being reported as an ordinary failure with an unusual status.

The function returns a non-nil error only for the fourth ending. This is a design decision with a cost. A caller that tests only `err` will conclude, wrongly, that a program that ran and failed was a success of the tool, and must also inspect the outcome in the result. The alternative, returning an error for every ending other than success, would have made the signature simpler and the information coarser, and since the ledger needs to record the

distinctions, `Run` reports them in the value and reserves the error for the case in which there is no meaningful value to report. Any such decision should be documented at the function, and the comment above `Run` states it.

5.3 Time and Cancellation

The timeout is implemented with the `context` package, which Part III develops at length. A context is a value that carries a deadline and a signal for cancellation across the boundaries of functions and goroutines. The call `context.WithTimeout` derives a context that is cancelled when the given duration has elapsed, and `exec.CommandContext` ties a command to it, so that when the context is done the program is killed. The deferred call to the `cancel` function releases the resources of the context as soon as `Run` returns, whether the command finished or not.

The tests for the runner revealed a subtlety that is easy to miss. The timeout test runs the command `sh -c "sleep 5"` with a limit of a tenth of a second. When the context expires, the kill signal is delivered to the shell, which is the process that was started. The `sleep` that the shell in turn started is a child of the shell and is not signalled. It still holds the write ends of the pipes through which the output of the shell is being captured, and the standard library's `Wait` does not return until those pipes are closed. In the first version of the runner, therefore, the test took the full five seconds, because the call was waiting for a process that the tool had already given up on. The remedy is the field `WaitDelay`, added to `exec.Cmd` in Go 1.20, which bounds how long `Wait` will wait for the pipes after the context is done. The runner sets it to half a second, and the test now finishes in a fraction of a second. The episode is a reminder that killing a process is not the same as killing the work it started, and that a tool which supervises programs must decide what happens to descendants. Part V returns to the question in the setting of containers, where the isolation of process groups and namespaces gives the tool a means of terminating a whole tree.

```
package runner

import (
    "strings"
    "testing"
    "time"
)

func TestSucceeded(t *testing.T) {
    res, err := Run(".", 5*time.Second, "sh", "-c", "echo hello")
    if err != nil {
        t.Fatal(err)
    }
    if res.Outcome != Succeeded || strings.TrimSpace(res.Stdout) != "
    hello" {
        t.Fatalf("%+v", res)
    }
}

```

```
func TestFailedKeepsExitCode(t *testing.T) {
    res, err := Run(".", 5*time.Second, "sh", "-c", "echo oops >&2; exit 3")
    if err != nil {
        t.Fatal(err)
    }
    if res.Outcome != Failed || res.ExitCode != 3 || !strings.Contains(res.Stderr, "oops") {
        t.Fatalf("%+v", res)
    }
}

func TestTimedOut(t *testing.T) {
    res, err := Run(".", 100*time.Millisecond, "sh", "-c", "sleep 5")
    if err != nil {
        t.Fatal(err)
    }
    if res.Outcome != TimedOut {
        t.Fatalf("%+v", res)
    }
}

func TestDidNotStart(t *testing.T) {
    res, err := Run(".", 5*time.Second, "definitely-not-a-program-golab")
    if err == nil || res.Outcome != NotStated {
        t.Fatalf("res=%+v err=%v", res, err)
    }
}
```

Listing 5.2: Tests for the four endings.

The tests in Listing 5.2 exercise each outcome in turn. They use `sh` as a convenient stand-in for any program, which ties them to systems that provide a POSIX shell. That dependency is acceptable for a tool developed in a Linux environment, and it is noted here so that it is a decision and not an oversight.

5.4 A Shell Pipeline and Its Go Counterpart

It is instructive to ask what the shell provides that the `runner` package does not. A pipeline such as `pdflatex paper.tex | tee build.log` connects the output of one program to the input of another, and the shell arranges the connection. In Go the connection must be made explicitly, either by assigning the `StdoutPipe` of one command to the `Stdin` of the next or by passing data through an `io.Pipe`. The explicitness is a burden for a one-line command and a benefit for a pipeline whose stages must be supervised separately, since each stage then has its own exit status, its own captured error output, and its own timeout, and the failure of one is visible instead of being masked by the status of the last. The shell offers the option `pipefail` to approximate this, and Go makes it the default shape of the code.

The bounded character of the `runner` package is deliberate. It runs one program, waits for it, and reports what happened. Chaining, retrying, and running programs in parallel are operations on top of it, and Part III builds them from goroutines and channels so that the supervision of a single program, once settled, is not reopened.

Exercises

Exercise 5.1. Write a test that runs a program with an argument containing spaces, quotation marks, and a dollar sign, and shows that the program receives it unchanged. Then write the corresponding Bash command and describe what quoting is required to achieve the same result.

Exercise 5.2. Remove the assignment to `WaitDelay` and run the timeout test. Measure how long it takes and explain the result in terms of the processes and pipes involved. Then change the test command to `exec sleep 5` and explain why the behaviour differs.

Exercise 5.3. Extend `Result` so that it also reports the signal that terminated a process, where there was one, and add a fifth outcome for a program killed by a signal other than the tool’s own. State what information the operator gains from the distinction.

Exercise 5.4. Implement a function that runs two programs as a pipeline, connecting the output of the first to the input of the second, and returns a separate result for each. Decide how the function should behave when the second program exits before the first has finished writing.

Chapter 6

A Ledger of Actions

6.1 What a Ledger Is For

A tool that finds files, hashes them, and runs programs on them produces consequences, and an operator who returns to the tool after a week will want to know what it did. Terminal output is gone by then, and the state of the filesystem records only the net effect of the actions and not the sequence. A log file is the usual remedy, but an ordinary log has a weakness that matters in a setting where the point of keeping records is to rely on them later: any line can be changed or removed, and nothing in the file reveals that the change took place. The ledger of `golab` is an append-only sequence of records in which each record includes the digest of the record before it, so that a change to an earlier line breaks the chain and can be found by recomputation.

The ledger records events and does not judge them, and it is worth being exact about what that sentence covers. In Part I the ledger receives one record for each `verify` and each `run` command, and nothing for `discover`, which is read-only. A `verify` record holds counts and not per-file results. No record names the person or process that issued the command, since the tool has no notion of identity. The ledger also does not accept every write unconditionally: `Append` refuses to extend a chain that already fails the check, which is a refusal made by the ledger itself and is deliberate. It states that a verification was performed, that a program was run and ended in a certain way, and that these happened in this order, and it takes no position on whether they were permitted. That separation between recording and authorizing is one of the central distinctions of the book, and in Part I it is observed by construction, since the program has no policy beyond a few classification rules and the admissibility checks of the transition package. Part II introduces policy as a separate component and then raises the question of what a ledger of events should say about refusals.

6.2 The Record and Its Digest

A record has a sequence number, a timestamp, an action, a subject, a map of details, the digest of its predecessor, and its own digest. The structure is declared with field tags that control its encoding as JSON, and each record occupies one line of a file, a format

commonly called JSON Lines, which has the convenient property that appending a record never requires rewriting what is already there.

```
// Package ledger keeps an append-only record of what the tool did. Each
// record carries the digest of its predecessor, so any later alteration of
// an earlier line is detectable. The ledger records events; it does not
// decide whether they were permitted.
package ledger

import (
    "bufio"
    "crypto/sha256"
    "encoding/hex"
    "encoding/json"
    "errors"
    "fmt"
    "io"
    "os"
    "time"
)

// Record is one line of the ledger.
type Record struct {
    Seq     int               `json:"seq"`
    Time    time.Time         `json:"time"`
    Action  string            `json:"action"`
    Subject string            `json:"subject"`
    Detail  map[string]string `json:"detail,omitempty"`
    Prev    string            `json:"prev"`
    Digest  string            `json:"digest"`
}

// ErrBroken reports that the chain of digests does not hold.
var ErrBroken = errors.New("ledger chain broken")

// MaxLine is the longest encoded record, newline excluded, that Load can
// read back. Append refuses a longer one, because a record that can be
// written but not read would make every later append fail.
const MaxLine = 1 << 20

// ErrTooLarge reports a record whose encoding exceeds MaxLine.
var ErrTooLarge = errors.New("ledger record too large")

// digest computes the record digest over every field except Digest itself.
```

```
// json.Marshal sorts map keys, so the encoding is deterministic.
func digest(r Record) (string, error) {
    r.Digest = ""
    b, err := json.Marshal(r)
    if err != nil {
        return "", err
    }
    sum := sha256.Sum256(b)
    return hex.EncodeToString(sum[:]), nil
}

// Load reads every record from path. A missing file is an empty ledger.
func Load(path string) ([]Record, error) {
    f, err := os.Open(path)
    if errors.Is(err, os.ErrNotExist) {
        return nil, nil
    }
    if err != nil {
        return nil, fmt.Errorf("open ledger: %w", err)
    }
    defer f.Close()

    var records []Record
    sc := bufio.NewScanner(f)
    sc.Buffer(make([]byte, 0, 64*1024), MaxLine+1)
    for line := 1; sc.Scan(); line++ {
        var r Record
        if err := json.Unmarshal(sc.Bytes(), &r); err != nil {
            return nil, fmt.Errorf("ledger line %d: %w", line, err)
        }
        records = append(records, r)
    }
    if err := sc.Err(); err != nil {
        return nil, fmt.Errorf("read ledger: %w", err)
    }
    return records, nil
}

// Check walks the chain and returns ErrBroken, wrapped with the position,
// at the first record that does not match.
func Check(records []Record) error {
    prev := ""
    for i, r := range records {
        if r.Seq != i+1 {
            return fmt.Errorf("record %d has seq %d: %w", i+1, r.Seq, ErrBroken)
        }
        if r.Prev != prev {
            return fmt.Errorf("record %d has wrong predecessor: %w",
```

```
    r.Seq, ErrBroken)
        }
        want, err := digest(r)
        if err != nil {
            return err
        }
        if r.Digest != want {
            return fmt.Errorf("record %d digest mismatch: %w", r.Seq,
     ErrBroken)
        }
        prev = r.Digest
    }
    return nil
}

// Extend computes the record that would follow the existing ones. It
     does no
// input or output and does not check the chain; Append does both
    around it.
// The experiments of Part VI use it to build ledgers in memory.
func Extend(existing []Record, action, subject string, detail map[
    string]string, now time.Time) (Record, error) {
    r := Record{
        Seq:     len(existing) + 1,
        Time:    now.UTC().Truncate(time.Second),
        Action:  action,
        Subject: subject,
        Detail:  detail,
    }
    if n := len(existing); n > 0 {
        r.Prev = existing[n-1].Digest
    }
    var err error
    if r.Digest, err = digest(r); err != nil {
        return Record{}, err
    }
    return r, nil
}

// appendFile is the part of *os.File that writing a record uses. It
    exists
// so that tests can substitute a file that fails in chosen ways.
type appendFile interface {
    Write([]byte) (int, error)
    Stat() (os.FileInfo, error)
    Truncate(int64) error
    Sync() error
    Close() error
}

// openAppend opens the ledger for appending. Tests replace it;
```

```
    nothing else
// should. Because it is a variable, tests that replace it must not run in
// parallel with one another.
var openAppend = func(path string) (appendFile, error) {
    return os.OpenFile(path, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0o644)
}

// Append adds one record to the ledger at path. It refuses to extend a
// ledger whose existing chain is already broken.
//
// If the write fails part-way, as it does when the disk fills, Append cuts
// the file back to the length it had before, so a failed append leaves no
// fragment behind. After a failure of the final flush to stable storage the
// record may or may not survive a crash: an error from Append means that the
// record is not guaranteed to be present, not that it is absent.
func Append(path, action, subject string, detail map[string]string, now time.Time) (Record, error) {
    existing, err := Load(path)
    if err != nil {
        return Record{}, err
    }
    if err := Check(existing); err != nil {
        return Record{}, fmt.Errorf("refusing to append: %w", err)
    }
    r, err := Extend(existing, action, subject, detail, now)
    if err != nil {
        return Record{}, err
    }
    line, err := json.Marshal(r)
    if err != nil {
        return Record{}, err
    }
    if len(line) > MaxLine {
        return Record{}, fmt.Errorf("record of %d bytes exceeds %d: %w", len(line), MaxLine, ErrTooLarge)
    }
    if err := writeLine(path, append(line, '\n')); err != nil {
        return Record{}, err
    }
    return r, nil
}

// writeLine appends one line and flushes it, restoring the previous length
```

```
// of the file if the write does not complete.
func writeLine(path string, line []byte) error {
    f, err := openAppend(path)
    if err != nil {
        return fmt.Errorf("open ledger for append: %w", err)
    }
    st, err := f.Stat()
    if err != nil {
        f.Close()
        return fmt.Errorf("stat ledger: %w", err)
    }
    if n, werr := f.Write(line); werr != nil || n != len(line) {
        if werr == nil {
            werr = io.ErrShortWrite
        }
        err := fmt.Errorf("write ledger: %w", werr)
        if terr := f.Truncate(st.Size()); terr != nil {
            err = fmt.Errorf("%w; and the partial record could not be removed: %v", err, terr)
        }
        f.Close()
        return err
    }
    if err := f.Sync(); err != nil {
        f.Close()
        return fmt.Errorf("sync ledger: %w", err)
    }
    if err := f.Close(); err != nil {
        return fmt.Errorf("close ledger: %w", err)
    }
    return nil
}
```

Listing 6.1: The ledger package.

Let the records be $r_1, r_2, \ldots, r_N$ and let enc denote the JSON encoding of a record with its digest field set to the empty string. The digest of record n is defined by

$$d_n = H\big(\mathrm{enc}(r_n)\big), \qquad d_0 = \varepsilon,$$

where H is SHA-256 and ε the empty string, and the field of r_n that records its predecessor holds d_{n-1}. Because d_{n-1} is part of the encoded content of r_n, the value d_n depends on every earlier record, through the chain of predecessors. The function `digest` in Listing 6.1 computes the definition directly. It relies on a property of the JSON encoder that the documentation guarantees, namely that the keys of a map are written in sorted order and that the fields of a struct are written in the order of their declaration, so that the same record always encodes to the same bytes. Without that determinism the digest would not be reproducible, and a ledger written on one machine could not be checked on another.

A change to a record is reported at that record’s own sequence number, because its recomputed digest no longer agrees with the stored one, and the linkage test at the next record

is reached only if the digest was rewritten. The function `Check` verifies three things for each record in turn: that its sequence number equals its position, that its predecessor field equals the digest of the preceding record, and that its own digest equals the recomputed value. It reports the first failure, wrapped around the sentinel `ErrBroken`, and the wrapping carries the sequence number so that an operator is told where to look. The function `Append` loads the existing ledger, checks it, and refuses to extend one that fails the check, because appending to a broken chain would give the damage a respectable appearance. The new record is written with the file opened in append mode, and the error of the final `Close` is examined, since a failure to flush a write is reported there and not at the call to `Write`. The listing is the final form of the package, which Part VI revised in two ways that this chapter does not discuss: a write that fails part-way is cut back so that no fragment of a record remains, and the record is flushed to stable storage before the function returns. Chapter 20 gives the reasons.

6.3 What the Chain Establishes

The guarantee that the chain gives is narrower than it can appear, and it is worth stating exactly.

Proposition 6.1. *Suppose a ledger of N records satisfies `Check`, and suppose H is collision resistant. If a single record r_k with $k < N$ is replaced by a different record while all others are unchanged, then `Check` fails. If a trusted party holds the head digest d_N, then any ledger that satisfies `Check` and has head digest d_N has the same records as the original, in the same order.*

Proof. Let r'_k be the replacement for r_k with $k < N$. If r'_k keeps the old digest field, then the digest recomputed from its content differs from the stored one unless the two records encode identically, which contradicts their being different. If r'_k stores a digest recomputed from its own content, that digest differs from d_k by collision resistance, and record r_{k+1} still stores d_k as its predecessor, so the linkage test fails at $k+1$. For the second claim, a ledger that satisfies `Check` is determined by its records, and each digest is a function of everything before it, so two such ledgers with the same d_N must agree record by record, or else a collision exists. □

The proposition has two limits that follow from its own statement. The first is that the check alone does not protect the final record or the length of the ledger. A person who can rewrite the file can alter the last record and recompute its digest, or can delete trailing records, and the shortened file will still satisfy `Check`. The second is that the check establishes internal consistency and nothing about authenticity: an adversary who controls the whole file can produce a different ledger with perfectly consistent digests from beginning to end. The protection against both is the one the proposition names, which is a copy of the head digest held somewhere the adversary cannot reach, such as a printed note, a message sent to another party, or a record in a different system. The ledger is therefore best described as tamper-evident relative to an external anchor. It is not tamper-proof, and the version in Part I does not provide the anchor. The tool does not yet have a command that prints the head digest, and the first exercise adds one. Where the digest is stored is the operator’s decision, and the examples above are illustrations and not features of `golab`.

Two further limits belong to the implementation. Timestamps come from the clock of the machine, are truncated to whole seconds, and are not authenticated, so the ledger shows the order in which events were recorded and not trustworthy times. The subject of a run record is the program and its arguments joined with spaces, which loses the original quoting, as the transcript below shows, and is therefore a description for a human reader and not a command that can be re-executed.

```
package ledger

import (
    "errors"
    "os"
    "path/filepath"
    "strings"
    "testing"
    "time"
)

var t0 = time.Date(2026, 10, 25, 12, 0, 0, 0, time.UTC)

func TestAppendBuildsChain(t *testing.T) {
    path := filepath.Join(t.TempDir(), "ledger.jsonl")
    for i, action := range []string{"discover", "verify", "run"} {
        r, err := Append(path, action, "papers", map[string]string{"n
    ": "1"}, t0.Add(time.Duration(i)*time.Second))
        if err != nil {
            t.Fatal(err)
        }
        if r.Seq != i+1 {
            t.Fatalf("seq %d, want %d", r.Seq, i+1)
        }
    }
    records, err := Load(path)
    if err != nil {
        t.Fatal(err)
    }
    if len(records) != 3 {
        t.Fatalf("loaded %d records", len(records))
    }
    if err := Check(records); err != nil {
        t.Fatalf("chain should hold: %v", err)
    }
}

func TestTamperingIsDetected(t *testing.T) {
    path := filepath.Join(t.TempDir(), "ledger.jsonl")
    for _, a := range []string{"discover", "verify"} {
        if _, err := Append(path, a, "papers", nil, t0); err != nil {
            t.Fatal(err)
        }
```

```
    }
    raw, err := os.ReadFile(path)
    if err != nil {
        t.Fatal(err)
    }
    altered := strings.Replace(string(raw), `"discover"`, `"discoveR"`, 1)
    if altered == string(raw) {
        t.Fatal("test setup failed to alter the ledger")
    }
    if err := os.WriteFile(path, []byte(altered), 0o644); err != nil {
        t.Fatal(err)
    }

    records, err := Load(path)
    if err != nil {
        t.Fatal(err)
    }
    if err := Check(records); !errors.Is(err, ErrBroken) {
        t.Fatalf("want ErrBroken, got %v", err)
    }
    if _, err := Append(path, "run", "papers", nil, t0); !errors.Is(err, ErrBroken) {
        t.Fatalf("append to a broken ledger should fail with ErrBroken, got %v", err)
    }
}

func TestMissingLedgerIsEmpty(t *testing.T) {
    records, err := Load(filepath.Join(t.TempDir(), "none.jsonl"))
    if err != nil || len(records) != 0 {
        t.Fatalf("records=%v err=%v", records, err)
    }
}

func TestOversizedRecordIsRefusedAndLedgerStaysReadable(t *testing.T) {
    path := filepath.Join(t.TempDir(), "l.jsonl")
    now := time.Unix(1_700_000_000, 0)
    if _, err := Append(path, "a", "s", nil, now); err != nil {
        t.Fatal(err)
    }
    big := map[string]string{"blob": strings.Repeat("x", MaxLine)}
    if _, err := Append(path, "b", "s", big, now); !errors.Is(err, ErrTooLarge) {
        t.Fatalf("err = %v, want ErrTooLarge", err)
    }
    if _, err := Append(path, "c", "s", nil, now); err != nil {
        t.Fatalf("ledger unusable after refusal: %v", err)
    }
```

```
    rs, err := Load(path)
    if err != nil || len(rs) != 2 {
        t.Fatalf("len=%d err=%v", len(rs), err)
    }
}
```

Listing 6.2: Tests for the ledger chain.

The tests of Listing 6.2 check the three behaviours that matter. The first appends three records and confirms that the chain holds when reloaded. The second corrupts the file by changing one character of the first record and confirms that `Check` reports `ErrBroken` and that `Append` refuses to extend the damaged ledger. The third confirms that a missing file is read as an empty ledger, which is the state in which the first append creates the file.

6.4 The Command and a Recorded Session

The command-line program connects the pieces. Its `main` function does nothing except pass the arguments to a function named `run` and exit with the status that `run` returns, so that the decision about the process exit status is made in one place and the rest of the program is testable. The complete source is in Appendix A. Each subcommand follows the same shape, in which it performs an operation, appends a record of the outcome to the ledger, and only then returns an error if the operation failed, which is the ordering argued for in Chapter 4.

The following transcript is the output of an actual session. The directory `papers` contains a text file, a TEX file, and a summary of the second. The session discovers the files, verifies them, runs two programs of which one fails, audits the ledger, and then alters the first line of the ledger and audits again.

```
$ golab discover papers
source          3  papers/a.txt
summary         1  papers/b.summary.md
source         23  papers/b.tex
$ golab verify papers
ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad  papers
    /a.txt
2d711642b726b04401627ca9fbac32f5c8530fb1903cc4db02258717921a4881  papers
    /b.summary.md
9f1c4eec1f3f6ab16d74135508a6667bb4ec027155c506bad8dfca807899f0a8  papers
    /b.tex
$ golab run papers sh -c 'echo built'
built
[succeeded, exit 0, 2ms]
$ golab run papers sh -c 'exit 4'
[failed, exit 4, 2ms]
golab: program failed
(exit status 1)
$ golab audit
ledger holds: 3 records
```

```
$ cat golab-ledger.jsonl
{"seq":1,"time":"2026-10-25T16:56:21Z","action":"verify","subject":"
   papers","detail":{"failed":"0","found":"3","verified":"3"},"prev
   ":"","digest":"15846
   b57cae5f478de8360d3bf9256c2bcabb03b25ed3d2156e6c646dfe422a0"}
{"seq":2,"time":"2026-10-25T16:56:21Z","action":"run","subject":"sh -c
   echo built","detail":{"elapsed":"2ms","exit":"0","outcome":"
   succeeded"},"prev":"15846
   b57cae5f478de8360d3bf9256c2bcabb03b25ed3d2156e6c646dfe422a0","digest
   ":"a757eb56c05dcb6202a94f8f476b08d297e60a5fe35cfab0a7335c36230d6e11"}

{"seq":3,"time":"2026-10-25T16:56:21Z","action":"run","subject":"sh -c
   exit 4","detail":{"elapsed":"2ms","exit":"4","outcome":"failed"},"
   prev":"
   a757eb56c05dcb6202a94f8f476b08d297e60a5fe35cfab0a7335c36230d6e11","
   digest":"97
   bde7e4cb19c9f91a81abb308e785e9a1af9be0aef5fc579625421101b50dd3"}
$ sed -i '1s/verify/verifY/' golab-ledger.jsonl
$ golab audit
golab: record 1 digest mismatch: ledger chain broken
(exit status 1)
$ golab run papers sh -c 'echo again'
golab: refusing to append: record 1 digest mismatch: ledger chain broken
(exit status 1)
```

Listing 6.3: A recorded session of `golab`.

Several features of the transcript carry the argument of the chapter. The kinds in the first listing show the classification at work, with the summary distinguished from the sources. The digest of `a.txt` is the standard value for the three bytes `abc`. The failed program is recorded in the ledger with its outcome and exit status, and the command exits with status 1 so that a surrounding shell script can detect the failure. The audit reports that the chain holds until the first line is altered by changing a single letter, after which the audit fails and, more significantly, the next attempt to run a program is refused until the ledger is repaired. A tool that continued to extend a ledger it knew to be damaged would be undermining the one property that gives the ledger its use.

6.5 Where Part I Ends

The tool at this point can find files, state what it has read of them, run programs on them, and keep a chained record of what it did, and each of these capacities has a type or a package that names it. The four responsibilities of Chapter 1 have homes: the command in `cmd/golab`, the operations in `artifact` and `runner`, the record in `ledger`, and a rudimentary policy in the classification rules and the transition checks. What the tool lacks is also clear. It has no explicit policy that decides whether an operation is permitted, and the ledger cannot say that something was refused for reasons of policy. It cannot safely run two operations at once, since two processes that append at the same moment can each read the same predecessor and write records with the same sequence number, a fault that `Check`

would detect afterward and that nothing prevents. It does not anchor the head of its ledger anywhere outside the file. These absences are the agenda of the later parts of the book, and the plan for them follows this chapter.

Exercises

Exercise 6.1. Add a command `head` that prints the sequence number and digest of the last record, and write a test. Then describe a procedure by which an operator could use the output to detect the deletion of trailing records, and state what the procedure assumes about where the output is kept.

Exercise 6.2. Run two `golab run` commands at the same instant in a loop until the ledger is damaged, and report how the damage appears to `Check`. Propose a locking strategy based on a lock file, and identify a failure of that strategy.

Exercise 6.3. Extend the record so that a verification event includes the digests of the files it verified, and consider the size of the resulting record for a directory of ten thousand files. Discuss whether the digests belong in the ledger or in a separate manifest whose own digest is recorded in the ledger.

Exercise 6.4. Design, without implementing it, a record for a refusal, that is, for a command that was rejected before any operation was attempted. State which fields it needs, what the subject should be, and why the ledger of Part I has no honest way to produce it.

Part II

Governed Workflows

Chapter 7

Interfaces and Composition

7.1 Why Policy Is a Separate Thing

At the end of Part I the tool could do things and record that it had done them, but it had no way of saying that a thing should not be done. The only rules in the program were scattered through the code, as the classification of file names, the bounds of the counter, and the refusal to extend a broken ledger. A rule that lives inside the function it governs cannot be varied without editing the function, cannot be shown to an auditor without reading the function, and cannot be combined with other rules except by writing a larger function. The design question of Part II is how to move rules out of the operations and into values that can be named, passed around, tested on their own, and combined.

Go answers the question with the interface, and the answer is characteristic of the language. An interface in Go is a set of method signatures, and a type satisfies it by having those methods, with no declaration of intent anywhere. The standard library uses this heavily. The hash of Chapter 3 satisfied `io.Writer` without knowing the interface existed, and the buffers of Chapter 5 did the same. The consequence for design is that the consumer of a behaviour defines the interface it needs, in its own package, and any type anywhere that happens to have the right methods can be supplied. The package of this chapter defines the single interface through which every rule of `golab` is expressed. The interface has one method, named `Decide`, which takes a record and returns a verdict.

```
1  package pipeline
2
3  import "strings"
4
5  // Verdict is what a policy may say about a record. The values are
       ordered
6  // by severity, so that the most restrictive of several verdicts is
       simply
7  // the largest.
8  type Verdict int
9
10 const (
11     Allow Verdict = iota
```

```
    Unscoped
    Gate
    Quarantine
    Refuse
)

func (v Verdict) String() string {
    switch v {
    case Allow:
        return "allow"
    case Unscoped:
        return "unscoped"
    case Gate:
        return "gate"
    case Quarantine:
        return "quarantine"
    case Refuse:
        return "refuse"
    }
    return "unknown"
}

// valid reports whether v is one of the declared verdicts.
func (v Verdict) valid() bool { return v >= Allow && v <= Refuse }

func (v Verdict) disposition() Disposition {
    switch v {
    case Allow:
        return DispAdmitted
    case Unscoped:
        return DispUnscoped
    case Gate:
        return DispGated
    case Quarantine:
        return DispQuarantined
    default:
        // Refuse, and any value outside the declared set, fail closed.
        return DispRefused
    }
}

// Policy decides what verdict a record receives.
type Policy interface {
    Decide(Record) Verdict
}

// PolicyFunc adapts an ordinary function to the Policy interface.
type PolicyFunc func(Record) Verdict

// Decide calls f.
```

```
func (f PolicyFunc) Decide(r Record) Verdict { return f(r) }

// AllOf combines policies. The most restrictive verdict wins. With no
// policies at all it allows everything, so callers that want a default of
// refusal must say so.
func AllOf(ps ...Policy) Policy {
    return PolicyFunc(func(r Record) Verdict {
        worst := Allow
        for _, p := range ps {
            v := p.Decide(r)
            if !v.valid() {
                v = Refuse // an undeclared verdict fails closed, on either side of the range
            }
            if v > worst {
                worst = v
            }
        }
        return worst
    })
}

// DenyAuthors refuses records written by any of the named authors.
func DenyAuthors(names ...string) Policy {
    denied := make(map[string]bool, len(names))
    for _, n := range names {
        denied[n] = true
    }
    return PolicyFunc(func(r Record) Verdict {
        if denied[r.Author] {
            return Refuse
        }
        return Allow
    })
}

// RequireScope marks records outside the given scope prefix as unscoped.
func RequireScope(prefix string) Policy {
    return PolicyFunc(func(r Record) Verdict {
        if strings.HasPrefix(r.Scope, prefix) {
            return Allow
        }
        return Unscoped
    })
}

// GateContaining holds back records whose body contains the phrase, for
```

```
// review by someone outside the pipeline.
func GateContaining(phrase string) Policy {
    return PolicyFunc(func(r Record) Verdict {
        if strings.Contains(r.Body, phrase) {
            return Gate
        }
        return Allow
    })
}
```

Listing 7.1: Verdicts, the policy interface, and combinators.

7.2 Verdicts as an Ordered Vocabulary

A policy examines a record and returns a verdict, and the verdict is an integer type whose five constants are declared with the keyword `iota`. Within a constant declaration, `iota` starts at zero and increases by one on each line, so `Allow` is 0, `Unscoped` is 1, and so on up to `Refuse`, which is 4. The choice of an integer representation, in place of the string constants used for `Kind` and for the outcomes of the runner, is deliberate, because verdicts are ordered by severity and the order is used in the combinator below. The order is a fact about the vocabulary and is stated once, in the declaration, by the placement of the constants.

The method `String` on `Verdict` makes the values printable by name. The `fmt` package checks whether a value has a method with that signature, which is the interface `fmt.Stringer`, and uses it when formatting, so that a verdict in an error message appears as `refuse` and not as the number four. Again no declaration connects the type to the interface. The unexported method `disposition` maps each verdict to the disposition that an admitted record will receive, and it ends in a default branch that treats any unrecognized value as a refusal. A system that decides what is permitted should fail closed, meaning that when it meets something outside its vocabulary it denies and does not allow, and the branch is where that principle is applied to a conversion.

The reader will notice that the policy returns a `Verdict` and not a `Disposition`, although the two are closely related and the second has seven values to the first’s five. The difference is the point of the next chapter. Two of the seven dispositions, duplicate and malformed, are facts about the record that the pipeline determines for itself, and a policy has no business asserting them. By giving the policy a smaller vocabulary, the type system prevents a policy from claiming that a record is a duplicate, and the compiler enforces the restriction because a function that returns `Verdict` has no way to produce the other two.

7.3 Functions with Methods

Writing a new named type for every rule would be tedious, and Go provides a compact alternative that surprises readers coming from other languages. Any named type can have methods, including a named function type. The declaration of `PolicyFunc` says that a `PolicyFunc` is a function from a record to a verdict, and the method `Decide` on it

simply calls the function. The effect is an adapter: any function with the right signature can be converted to a `PolicyFunc` and thereby becomes a `Policy`. The standard library's `http.HandlerFunc` is the best-known instance of the idiom, and Part V will use it.

The constructors `DenyAuthors`, `RequireScope`, and `GateContaining` each return a policy built from a closure, a function that refers to variables of the function that created it. The closure in `DenyAuthors` captures a map of denied names, built once at construction and consulted on every call, so that the cost of building the set is paid when the policy is made and not each time a record is judged. The variables captured by a closure live as long as the closure does, and the garbage collector of Go makes that safe without the programmer managing it. One caution applies to every such construction. A closure that captures a map or slice shares it with whoever else holds it, so a policy that captured a map handed in by the caller would change behaviour if the caller later modified the map. The constructor here copies the names into a map of its own, and the habit of copying at the boundary appears again in the next chapter.

7.4 Composition and What It Preserves

The function `AllOf` combines any number of policies into one, and its rule is that the most restrictive verdict wins. Because the verdicts are ordered integers, the rule is the arithmetic maximum, and the implementation is a loop that keeps the largest value seen. An empty combination allows everything, which is a decision with consequences. A caller who builds a combination from a list that happens to be empty gets a policy that admits every record, and a system intended to deny by default would need to place an explicit refusal at the bottom of the list. The behaviour is documented at the function, and the command-line tool uses it deliberately, since running `golab admit` with no flags is a request to apply no rules.

The following proposition records what the combinator guarantees, and the guarantees are the reason the design can be relied on when policies are assembled from several sources.

Proposition 7.1. *Let $p_1, \dots, p_n$ be policies and let $p = \texttt{AllOf}(p_1, \dots, p_n)$. Then for every record r, $p(r) = \max\{\text{Allow}, \hat{p}_1(r), \dots, \hat{p}_n(r)\}$, where $\hat{p}_i(r)$ is $p_i(r)$ if that is a declared verdict and* Refuse *otherwise. Consequently (a) the result does not depend on the order in which the policies are listed or on any repetition among them, (b) for any further policy q, $\texttt{AllOf}(p_1, \dots, p_n, q)(r) \geq p(r)$, and (c) a record is admitted under p only if it is admitted under each p_i taken alone.*

Proof. The formula is the definition of the loop. Part (a) holds because the maximum of a finite set is unchanged by reordering or repeating its members. Part (b) holds because adding a member to a set cannot lower its maximum. For (c), a record is admitted exactly when its verdict is `Allow`, the lowest value, so $p(r) =$ Allow forces every $\hat{p}_i(r)$ to be `Allow`, and a declared $p_i(r)$ equal to `Allow` is then the verdict of p_i alone. □

The proposition is a statement about a join-semilattice in which the policies are combined by taking the maximum, and its practical content is that adding a rule to a combination can only make the system stricter. A reader who wants to know why a record was admitted need only check that every rule individually permits it. The property is checked by tests in the next chapter, which shuffle the order of the policies and confirm that the dispositions do

not change, and which add policies one at a time and confirm that the number of admitted records never rises.

The first draft of `AllOf` did not contain the clause about undeclared verdicts. Its loop began at `Allow` and kept any larger value, which silently discarded a verdict below `Allow`. A policy that returned -1 was refused when used alone, because the conversion to a disposition falls through to the refusing default, and was ignored when used inside `AllOf`, because the maximum began at zero. A verdict above `Refuse`, such as 99, never had this problem: it won the maximum and was then refused by the same default. The flaw was confined to values below `Allow`. The inconsistency was found while writing the proof above, when the formula for $p(r)$ could not be made to hold for all integer values, and the repair is the call to `valid` that sends any undeclared verdict to `Refuse`. The case shows a general hazard of integer vocabularies in a language without closed enumerations, and it is the reason the tests include verdicts outside the declared range on both sides.

Exercises

Exercise 7.1. Write a policy constructor `AnyOf` that returns the *least* restrictive verdict of its arguments. State the algebraic properties that correspond to those of Proposition 7.1, and explain why a system that combined independent authorities with `AnyOf` would be harder to audit than one that used `AllOf`.

Exercise 7.2. Define a policy as a struct with a method instead of a closure, for instance a `Denylist` type holding a map, and show that it satisfies `Policy`. Compare the two styles with respect to inspection: which makes it easier for a program to print the rules it is applying?

Exercise 7.3. Make `AllOf` with no arguments refuse everything, and identify every place in the module whose behaviour changes. Argue for or against the change.

Exercise 7.4. Demonstrate the aliasing hazard described in this chapter. Write a constructor that captures a caller’s slice without copying it, show a test in which later modification of the slice changes the behaviour of an already constructed policy, and then repair the constructor.

Chapter 8

Inscription and Validation

8.1 Three Questions That Are Easy to Merge

Chapter 1 observed that a single function can answer, without saying so, three different questions about a proposed change: whether it is well formed, whether the actor is entitled to request it, and whether it is made durable. The distinction is easy to lose because a small program answers all three in one place and nothing breaks. It matters as soon as a system must explain itself, because the three questions have different answers to the question of who is responsible. A record can be perfectly well formed and still not be permitted. A record can be permitted and not yet stored. A record can be stored, receipted, and preserved forever, and still never be treated as true.

The pipeline of this part separates the questions into stages and gives each stage a type. A record is first *inscribed*, which means stored and receipted, and which involves no judgment of any kind. The set of inscriptions is then *validated*, which checks the structure of the whole set without reference to anyone’s rules. The valid set is then *admitted* under a policy, which assigns each record a disposition. The admitted records are *projected* onto a surface, meaning that the records belonging to a given scope are selected, and the projection is *activated*, which makes it live provided that it is internally coherent. The five stages follow the staging described in the author’s specification of a public claim graph. The code of this chapter and the next is a teaching model of that staging and does not implement the specification: it has no signatures, no argument semantics, and no claim to the formal properties that the specification proves. Where the text speaks of a property, it means the property of this small program, checked by this program’s tests.

```
// Package pipeline is a teaching model of a five-stage record pipeline:
// inscribe, validate, admit, project, activate. Each stage consumes the
// type produced by the previous stage and produces a type of its own, so
// the compiler refuses to confuse a record that has merely been stored
// with one that has been admitted.
package pipeline
```

```
7
8  import (
9      "bufio"
10     "crypto/sha256"
11     "encoding/hex"
12     "encoding/json"
13     "fmt"
14     "io"
15     "sort"
16 )
17
18 // Record is the content that an author inscribes.
19 type Record struct {
20     ID     string   `json:"id"`
21     Author string   `json:"author"`
22     Scope  string   `json:"scope"`
23     Body   string   `json:"body"`
24     Refs   []string `json:"refs,omitempty"`
25 }
26
27 // Receipt is the SHA-256 digest of the canonical encoding of a
       Record.
28 type Receipt string
29
30 // Inscribed is a record that has been stored and receipted.
       Inscription
31 // makes no judgment of any kind about the record.
32 type Inscribed struct {
33     rec     Record
34     receipt Receipt
35 }
36
37 // Record returns a copy of the inscribed record.
38 func (i Inscribed) Record() Record { return copyRecord(i.rec) }
39
40 // Receipt returns the receipt issued at inscription.
41 func (i Inscribed) Receipt() Receipt { return i.receipt }
42
43 func copyRecord(r Record) Record {
44     r.Refs = append([]string(nil), r.Refs...)
45     return r
46 }
47
48 // Inscribe stores a record and returns its receipt. The function
       takes no
49 // policy and returns no error: whatever is inscribed is inscribed,
       and
50 // whether it is acceptable is decided by later stages.
51 func Inscribe(r Record) Inscribed {
52     r = copyRecord(r)
53     b, _ := json.Marshal(r) // a struct of strings and a string slice
```

```
        cannot fail to encode
    sum := sha256.Sum256(b)
    return Inscribed{rec: r, receipt: Receipt(hex.EncodeToString(sum
    [:]))}
}

// LoadJSONL reads one Record per non-empty line.
func LoadJSONL(r io.Reader) ([]Record, error) {
    var out []Record
    sc := bufio.NewScanner(r)
    sc.Buffer(make([]byte, 0, 64*1024), 1<<20)
    for line := 1; sc.Scan(); line++ {
        if len(sc.Bytes()) == 0 {
            continue
        }
        var rec Record
        if err := json.Unmarshal(sc.Bytes(), &rec); err != nil {
            return nil, fmt.Errorf("records line %d: %w", line, err)
        }
        out = append(out, rec)
    }
    if err := sc.Err(); err != nil {
        return nil, fmt.Errorf("read records: %w", err)
    }
    return out, nil
}

func sortedIDs(m map[string]Inscribed) []string {
    ids := make([]string, 0, len(m))
    for id := range m {
        ids = append(ids, id)
    }
    sort.Strings(ids)
    return ids
}
```

Listing 8.1: Records, receipts, and inscription.

8.2 Inscription

The pipeline of Part II is a separate chain of types from the artifact types of Part I. `Inscribe` takes a `Record`, which is a claim with an author and a body, and not a file; it returns an `Inscribed`, which is not a `Verified`; and it does not write to the ledger. The two chains meet only in the command, which records the outcome of an `admit` run as a ledger event. A `Record` carries an identifier, an author, a scope, a body, and a list of identifiers of records it refers to. The function `Inscribe` encodes the record as JSON, hashes the encoding with SHA-256, and returns an `Inscribed` value holding the record and the resulting receipt. The signature is the whole of its specification: it takes a record and returns an inscription, it

has no error result, and it has no policy parameter. A record with no author, no body, and no identifier is inscribed as readily as a perfect one. This is an intended property and not an oversight, and it expresses the principle that storing is not endorsing. An archive that refused to store a document it disapproved of would no longer be an archive, and a ledger that silently dropped malformed records would conceal exactly the events an auditor most wants to see.

The receipt is deterministic, since the encoding of a struct follows the order of its fields and the encoding of a string slice follows the order of its elements, and so the same record always yields the same receipt, on any machine. The function `copyRecord` makes a fresh copy of the `Refs` slice before the record is stored. The copy answers a hazard that is specific to Go’s slices. A slice is a small value that refers to an underlying array, so assigning a slice, or placing it in a struct, shares the array and does not duplicate it. Had `Inscribe` stored the record as given, a caller who later modified the slice it passed in would change the contents of an inscription after its receipt had been issued, and the receipt would no longer describe the record it was supposed to attest. The test `TestInscribeCopiesRefs` fixes the behaviour. The accessor `Record` copies on the way out for the same reason.

8.3 Validation

Validation examines a set of inscriptions as a whole and asks whether it is structurally sound. A sound set has three properties: every record has a non-empty identifier, no identifier is attached to two different contents, and the references among the records form a graph with no dangling references and no cycles. The listing below shows the validation function together with the later stages, which the next chapter takes up.

```
package pipeline

import (
    "errors"
    "fmt"
    "strings"
)

// Errors reported by validation and activation.
var (
    ErrInconsistent = errors.New("inconsistent identifiers")
    ErrDangling     = errors.New("reference to an unknown record")
    ErrCycle        = errors.New("reference cycle")
    ErrOrphaned     = errors.New("active record refers to an inactive
     record")
)

// Valid is a set of inscribed records with consistent identifiers
    and
// acyclic, closed references. Only Validate can produce a non-zero
    Valid.
type Valid struct {
```

```
    items []Inscribed // sorted by ID
}

// Validate checks the structure of a set of inscriptions. Its signature
// has no policy parameter, so validity cannot depend on policy.
func Validate(set []Inscribed) (Valid, error) {
    byID := make(map[string]Inscribed, len(set))
    for _, in := range set {
        id := in.rec.ID
        if id == "" {
            return Valid{}, fmt.Errorf("record with empty id: %w", ErrInconsistent)
        }
        if prev, ok := byID[id]; ok {
            if prev.receipt != in.receipt {
                return Valid{}, fmt.Errorf("id %q inscribed with two different contents: %w", id, ErrInconsistent)
            }
            continue // the same inscription repeated is idempotent
        }
        byID[id] = in
    }
    ids := sortedIDs(byID)
    for _, id := range ids {
        for _, ref := range byID[id].rec.Refs {
            if _, ok := byID[ref]; !ok {
                return Valid{}, fmt.Errorf("%q refers to %q: %w", id, ref, ErrDangling)
            }
        }
    }
    const (
        white = iota
        grey
        black
    )
    colour := make(map[string]int, len(ids))
    var visit func(id string) error
    visit = func(id string) error {
        colour[id] = grey
        for _, ref := range byID[id].rec.Refs {
            switch colour[ref] {
            case grey:
                return fmt.Errorf("%q and %q: %w", id, ref, ErrCycle)
            case white:
                if err := visit(ref); err != nil {
                    return err
                }
            }
        }
```

```
        colour[id] = black
        return nil
    }
    for _, id := range ids {
        if colour[id] == white {
            if err := visit(id); err != nil {
                return Valid{}, err
            }
        }
    }
    items := make([]Inscribed, 0, len(ids))
    for _, id := range ids {
        items = append(items, byID[id])
    }
    return Valid{items: items}, nil
}

// Len reports how many distinct records the set holds.
func (v Valid) Len() int { return len(v.items) }

// Disposition is the outcome of admission for one record.
type Disposition string

const (
    DispAdmitted    Disposition = "admitted"
    DispDuplicate   Disposition = "duplicate"
    DispMalformed   Disposition = "malformed"
    DispUnscoped    Disposition = "unscoped"
    DispQuarantined Disposition = "quarantined"
    DispGated       Disposition = "gated"
    DispRefused     Disposition = "refused"
)

// Judgment pairs a record with the disposition it received.
type Judgment struct {
    Record      Record
    Receipt     Receipt
    Disposition Disposition
}

// Admitted is the outcome of running a Valid set through a policy.
type Admitted struct {
    judgments []Judgment
}

// Judgments returns a copy of the judgments in identifier order.
func (a Admitted) Judgments() []Judgment {
    out := make([]Judgment, len(a.judgments))
    for i, j := range a.judgments {
        j.Record = copyRecord(j.Record)
        out[i] = j
```

```
    }
    return out
}

// Counts tallies dispositions.
func (a Admitted) Counts() map[Disposition]int {
    m := make(map[Disposition]int)
    for _, j := range a.judgments {
        m[j.Disposition]++
    }
    return m
}

// Admit applies the policy to every record of a valid set. Two checks are
// made by the pipeline itself and are not left to policy: a record without
// an author or body is malformed, and a record whose author, scope and
// body repeat those of an earlier record (in identifier order) is a
// duplicate. A policy can return only a Verdict, which has no way to
// express either of those dispositions.
func Admit(v Valid, p Policy) Admitted {
    seen := make(map[string]string)
    out := make([]Judgment, 0, len(v.items))
    for _, in := range v.items {
        r := in.rec
        j := Judgment{Record: copyRecord(r), Receipt: in.receipt}
        key := r.Author + "\x00" + r.Scope + "\x00" + r.Body
        switch {
        case r.Author == "" || r.Body == "":
            j.Disposition = DispMalformed
        case seen[key] != "":
            j.Disposition = DispDuplicate
        default:
            seen[key] = r.ID
            j.Disposition = p.Decide(copyRecord(r)).disposition()
        }
        out = append(out, j)
    }
    return Admitted{judgments: out}
}

// Projection is the set of admitted records visible on one surface.
type Projection struct {
    scope string
    items []Record
}

// Project selects the admitted records whose scope begins with scope
.
```

```
func Project(a Admitted, scope string) Projection {
    p := Projection{scope: scope}
    for _, j := range a.judgments {
        if j.Disposition == DispAdmitted && strings.HasPrefix(j.Record.Scope, scope) {
            p.items = append(p.items, copyRecord(j.Record))
        }
    }
    return p
}

// Active is a projection that has been made live.
type Active struct {
    scope string
    items []Record
}

// Scope returns the surface on which the records are active.
func (a Active) Scope() string { return a.scope }

// Records returns a copy of the active records.
func (a Active) Records() []Record {
    out := make([]Record, len(a.items))
    for i, r := range a.items {
        out[i] = copyRecord(r)
    }
    return out
}

// Activate makes a projection live, provided that no active record refers
// to a record that the projection left out. Projection can cut references
// that validation found intact, so this check cannot be done earlier.
func Activate(p Projection) (Active, error) {
    present := make(map[string]bool, len(p.items))
    for _, r := range p.items {
        present[r.ID] = true
    }
    for _, r := range p.items {
        for _, ref := range r.Refs {
            if !present[ref] {
                return Active{}, fmt.Errorf("%q refers to %q, which is not in scope %q: %w", r.ID, ref, p.scope, ErrOrphaned)
            }
        }
    }
    return Active{scope: p.scope, items: p.items}, nil
}
```

Listing 8.2: Validation, admission, projection, and activation.

The function `Validate` builds a map from identifiers to inscriptions and checks the first two properties as it goes. Inscribing the same record twice is treated as idempotent, since the receipts agree, and so a set can be assembled from overlapping sources without being rejected for containing a repeat. An identifier is therefore allowed to appear more than once, provided that every appearance has identical content. Two different records under one identifier are an inconsistency. The check for dangling references runs next and the check for cycles last, using a depth-first search in which each identifier is white while unvisited, grey while its references are being explored, and black once all of them are done. Meeting a grey identifier during the search means that a reference leads back to a record whose own references are still being followed, which is a cycle. The three-colour scheme is the standard method, and it runs in time proportional to the number of records and references.

A detail of the implementation deserves comment because it affects reproducibility. The iteration order of a Go map is deliberately randomized by the runtime, so a loop over the map would visit the records in a different order on each run, and a validation that reported the first error it found would report a different one from run to run. The function therefore collects the identifiers, sorts them, and iterates over the sorted slice, so the error returned for an invalid set is the same every time and the valid set is stored in identifier order. A reader who has written shell scripts that loop over a glob will recognize the concern, since there too the output order is a property that must be established and cannot be assumed.

8.4 Validity Does Not Mention Policy

The central claim of this chapter is that validation, and inscription before it, are independent of any policy. The claim is expressed in the signatures: `Inscribe` takes a record and `Validate` takes a slice of inscriptions, and neither takes a `Policy`. The compiler therefore guarantees that no policy can be passed to them, and a function that cannot receive a policy cannot vary its behaviour according to one.

Proposition 8.1. *For every finite set of records R and every pair of policies p and q, the receipts produced by inscribing R and the result of validating the inscriptions are the same whether the set is subsequently admitted under p or under q.*

Proof. Inscription and validation are functions of the records alone. Admission is applied to the output of validation and is not an input to it, and no stage feeds a policy’s result back to an earlier stage. Hence the receipts and the validity of R are determined before either policy is consulted. □

The proof is short because the design makes it short, and the honesty of the claim depends on a qualification that the language forces. Go does not enforce purity. A function that takes no policy parameter could still read a package-level variable that holds a policy, or the clock, or a file, and the signature would not reveal it. The compiler’s guarantee is the narrow one that no policy is passed in, and the broader statement that nothing

else influences the result rests on the code of the two functions, which can be read in a minute, and on the test `TestPolicyDoesNotAffectInscriptionOrValidity`. The test builds the same records, runs them through admission under three quite different policies, and checks that the receipts and the size of the valid set are identical each time. It checks instances of the proposition and does not prove it. The distinction between an argument that covers all cases and a test that covers some is one the book keeps in view throughout.

8.5 The Zero Value, Again

The type `Valid` follows the pattern of `Verified` in having unexported fields, so that only `Validate` can build a non-empty one. Its zero value, however, differs in an instructive way. The zero `Verified` claimed to have read a file and had not, so it was dangerous and needed an `IsZero` guard. The zero `Valid` is an empty set of records, and an empty set satisfies every structural property vacuously, so it is a legitimate valid set and passing it on is harmless. A type whose zero value is meaningful and safe needs no guard, and designing types so that the zero value is a sensible state is an old Go proverb that this contrast makes concrete.

Exercises

Exercise 8.1. Change `Validate` so that a reference to an unknown record is allowed and treated as a pending reference. Identify which later stage must then change, and what the proposition on policy independence still guarantees.

Exercise 8.2. Remove the sort from `Validate` and run the tests repeatedly with the `-count` flag. Report which tests become unreliable and explain why in terms of map iteration.

Exercise 8.3. The receipt covers the record but not the time at which it was inscribed. Add an inscription time without breaking the determinism of receipts, and decide whether the time should be part of the receipt.

Exercise 8.4. Write a function that, given a valid set, returns the records in an order such that every record follows the records it refers to. Explain why such an order exists exactly because the set is acyclic.

Chapter 9

Admission, Projection, and Refusal

9.1 Admission

Admission is the first stage at which a policy is consulted, and it is where a valid set of records receives its dispositions. The function `Admit`, in Listing 8.2, visits the records in identifier order and assigns each one of seven dispositions. Two of them are determined by the pipeline itself. A record with no author or no body is malformed, and a record whose author, scope, and body repeat those of an earlier record in identifier order is a duplicate. Only a record that is neither is shown to the policy, and the policy's verdict is converted to one of the remaining five dispositions: admitted, unscoped, gated, quarantined, or refused.

The ordering of these tests encodes a decision. Malformed records are never shown to a policy, so a policy is never asked to reason about a record that lacks the fields it would inspect, and a policy written without defensive checks cannot be made to fail by a record with an empty body. A duplicate is likewise detected before the policy runs, so the policy is not asked to judge the same content twice. The consequence to be aware of is that duplicate detection depends on the order of identifiers, since the earlier record in that order is the original and the later is the repeat. Two records with identical content but different identifiers are therefore treated asymmetrically, and an author who wishes a particular one to count as the original must choose identifiers accordingly. The behaviour is deterministic and documented, and it is a convention of this model and not a law.

A record that the policy refuses still registers as seen, so that a later identical record is reported as a duplicate of a refused record. This is a small instance of a general question for any system that keeps state about what it has judged: whether the memory of a judgment should include the judgments that went against the subject. The choice here is the simpler one, and the first exercise asks for the alternative.

9.2 Seven Dispositions, Five Verdicts

The split between a seven-valued disposition and a five-valued verdict deserves a second look, because it is an example of using types to divide authority. The pipeline holds authority over what is structurally wrong or redundant, and the policy holds authority over what is permitted, and each is confined to its own vocabulary. A policy author cannot

declare a record malformed in order to hide it, since the type of the policy’s result has no such value, and the pipeline’s author cannot make a record refused for policy reasons without a policy being consulted. The separation does not decide who deserves to hold either authority. It makes the boundary visible in the types, so that anyone who wishes to move it has to change a declaration that is seen in review and cannot do so by an incidental edit to a function body.

9.3 Projection and Activation

A projection selects the admitted records visible on one surface, where a surface is identified by a scope prefix. The function `Project` takes only records whose disposition is admitted and whose scope begins with the given prefix, and it copies them so that the projection owns its data. Activation then makes the projection live, subject to one check that cannot be done earlier. A projected record may refer to a record that the projection left out, either because that record belongs to another scope or because it was not admitted, and an active record whose foundations are invisible on its own surface is an orphan. The function `Activate` returns `ErrOrphaned` if any projected record has such a reference. The check is a single pass over the direct references of the projected records and is not recursive, because if every direct reference of every projected record lies inside the projection, then so does every indirect one.

The need for the check shows why the stages are distinct. Validation found every reference intact, since the full set was closed and acyclic. Admission may then have refused or gated some of the referred-to records, and projection may have separated the rest by scope, and the coherence of the set was destroyed by operations that were each individually reasonable. No earlier stage could have detected this, because the damage did not exist until later stages acted. A system that validated once at the start and never again would present orphans as though they were sound.

```
package pipeline

import (
    "errors"
    "reflect"
    "strings"
    "testing"
)

func rec(id, author, scope, body string, refs ...string) Record {
    return Record{ID: id, Author: author, Scope: scope, Body: body,
    Refs: refs}
}

func inscribeAll(rs ...Record) []Inscribed {
    out := make([]Inscribed, len(rs))
    for i, r := range rs {
        out[i] = Inscribe(r)
    }
```

```
    return out
}

func TestInscriptionIgnoresContentQuality(t *testing.T) {
    bad := rec("", "", "", "")
    in := Inscribe(bad)
    if in.Receipt() == "" {
        t.Fatal("even a malformed record receives a receipt")
    }
    if Inscribe(bad).Receipt() != in.Receipt() {
        t.Fatal("receipts must be deterministic")
    }
}

func TestInscribeCopiesRefs(t *testing.T) {
    refs := []string{"a"}
    in := Inscribe(rec("b", "x", "s", "t", refs...))
    refs[0] = "changed"
    if in.Record().Refs[0] != "a" {
        t.Fatal("inscription shares memory with the caller")
    }
}

func TestValidate(t *testing.T) {
    cases := []struct {
        name string
        set  []Inscribed
        err  error
    }{
        {"ok", inscribeAll(rec("a", "x", "s", "t"), rec("b", "x", "s", "t2", "a")), nil},
        {"empty id", inscribeAll(rec("", "x", "s", "t")), ErrInconsistent},
        {"conflicting id", inscribeAll(rec("a", "x", "s", "t"), rec("a", "x", "s", "other")), ErrInconsistent},
        {"repeat is idempotent", inscribeAll(rec("a", "x", "s", "t"), rec("a", "x", "s", "t")), nil},
        {"dangling", inscribeAll(rec("a", "x", "s", "t", "zzz")), ErrDangling},
        {"self cycle", inscribeAll(rec("a", "x", "s", "t", "a")), ErrCycle},
        {"two cycle", inscribeAll(rec("a", "x", "s", "t", "b"), rec("b", "x", "s", "u", "a")), ErrCycle},
    }
    for _, c := range cases {
        t.Run(c.name, func(t *testing.T) {
            _, err := Validate(c.set)
            if !errors.Is(err, c.err) {
                t.Fatalf("err = %v, want %v", err, c.err)
            }
        })
```

```
    }
}

func sample() Valid {
    v, err := Validate(inscribeAll(
        rec("r1", "ann", "lab/a", "claim one"),
        rec("r2", "bob", "lab/a", "claim two", "r1"),
        rec("r3", "bob", "lab/b", "claim three"),
        rec("r4", "ann", "lab/a", "claim one"), // repeats r1
        rec("r5", "", "lab/a", "no author"),
        rec("r6", "cy", "lab/a", "needs REVIEW before use"),
    ))
    if err != nil {
        panic(err)
    }
    return v
}

func dispositions(a Admitted) map[string]Disposition {
    m := map[string]Disposition{}
    for _, j := range a.Judgments() {
        m[j.Record.ID] = j.Disposition
    }
    return m
}

func TestAdmitDispositions(t *testing.T) {
    p := AllOf(DenyAuthors("bob"), RequireScope("lab/a"),
    GateContaining("REVIEW"))
    got := dispositions(Admit(sample(), p))
    want := map[string]Disposition{
        "r1": DispAdmitted,
        "r2": DispRefused,
        "r3": DispRefused, // refused outranks unscoped
        "r4": DispDuplicate,
        "r5": DispMalformed,
        "r6": DispGated,
    }
    if !reflect.DeepEqual(got, want) {
        t.Fatalf("got %v\nwant %v", got, want)
    }
}

func TestPolicyDoesNotAffectInscriptionOrValidity(t *testing.T) {
    rs := []Record{rec("a", "x", "s", "t"), rec("b", "y", "s", "u", "
    a")}
    policies := []Policy{
        AllOf(),
        DenyAuthors("x", "y"),
        PolicyFunc(func(Record) Verdict { return Quarantine }),
    }
```

```
    var receipts [][]Receipt
    var lens []int
    for _, p := range policies {
        set := inscribeAll(rs...)
        v, err := Validate(set)
        if err != nil {
            t.Fatal(err)
        }
        _ = Admit(v, p) // policy enters only here
        var rr []Receipt
        for _, in := range set {
            rr = append(rr, in.Receipt())
        }
        receipts = append(receipts, rr)
        lens = append(lens, v.Len())
    }
    for i := 1; i < len(policies); i++ {
        if !reflect.DeepEqual(receipts[0], receipts[i]) || lens[0] != lens[i] {
            t.Fatalf("policy %d changed receipts or validity", i)
        }
    }
}

func TestUnknownVerdictFailsClosed(t *testing.T) {
    for _, bad := range []Verdict{99, -1} {
        raw := PolicyFunc(func(Record) Verdict { return bad })
        for name, p := range map[string]Policy{"direct": raw, "combined": AllOf(raw, AllOf())} {
            a := Admit(sample(), p)
            if a.Counts()[DispAdmitted] != 0 {
                t.Fatalf("verdict %d (%s) admitted something: %v", bad, name, a.Counts())
            }
        }
    }
}

func TestAllOfIsOrderIndependent(t *testing.T) {
    a, b, c := DenyAuthors("bob"), RequireScope("lab/a"), GateContaining("REVIEW")
    orders := [][]Policy{{a, b, c}, {c, b, a}, {b, a, c}, {a, a, b, c}}
    var first map[string]Disposition
    for i, ps := range orders {
        got := dispositions(Admit(sample(), AllOf(ps...)))
        if i == 0 {
            first = got
        } else if !reflect.DeepEqual(first, got) {
            t.Fatalf("order %d changed the outcome", i)
        }
```

```
    }
}

func TestAddingAPolicyNeverAdmitsMore(t *testing.T) {
    base := AllOf(RequireScope("lab/"))
    more := AllOf(RequireScope("lab/"), DenyAuthors("ann"),
    GateContaining("claim two"))
    nb := Admit(sample(), base).Counts()[DispAdmitted]
    nm := Admit(sample(), more).Counts()[DispAdmitted]
    if nm > nb {
        t.Fatalf("adding policies admitted more: %d > %d", nm, nb)
    }
}

func TestActivateDetectsOrphans(t *testing.T) {
    // r2 refers to r1; project only a scope that contains r2 but not
     r1.
    v, err := Validate(inscribeAll(
        rec("r1", "ann", "lab/a", "base"),
        rec("r2", "ann", "lab/b", "built on base", "r1"),
    ))
    if err != nil {
        t.Fatal(err)
    }
    a := Admit(v, AllOf())
    if _, err := Activate(Project(a, "lab/b")); !errors.Is(err,
    ErrOrphaned) {
        t.Fatalf("want ErrOrphaned, got %v", err)
    }
    act, err := Activate(Project(a, "lab/"))
    if err != nil || len(act.Records()) != 2 {
        t.Fatalf("err=%v records=%v", err, act.Records())
    }
}

func TestLoadJSONL(t *testing.T) {
    in := `{"id":"a","author":"x","scope":"s","body":"t"}` + "\n\n" +
        `{"id":"b","author":"y","scope":"s","body":"u","refs":["a"]}`
     + "\n"
    got, err := LoadJSONL(strings.NewReader(in))
    if err != nil || len(got) != 2 || got[1].Refs[0] != "a" {
        t.Fatalf("got %+v err %v", got, err)
    }
    if _, err := LoadJSONL(strings.NewReader("{bad")); err == nil {
        t.Fatal("malformed JSON must be an error")
    }
}
```

Listing 9.1: Tests for the pipeline.

9.4 What the Tests Check

The tests in Listing 9.1 are organized around the claims of the last three chapters. The table in `TestValidate` covers the success case and each failure the validator can report, including the repeat that is accepted as idempotent and both the self-cycle and the two-record cycle. The test `TestAdmitDispositions` builds a set of six records, chosen so that five of the seven dispositions are exercised (the unscoped disposition appears in the session below, and the quarantine only in the combinator tests), and compares the result with the expected dispositions in full. The record refused for its author and also outside the required scope is expected to be refused, which checks that the more severe verdict prevails. The tests `TestAllOfIsOrderIndependent` and `TestAddingAPolicyNeverAdmitsMore` check parts (a) and (b) of Proposition 7.1 on instances, and `TestUnknownVerdictFailsClosed` checks both the direct and the combined path for verdicts outside the declared range. Finally `TestActivateDetectsOrphans` builds the situation described above, in which one record in one scope refers to a record in another, and confirms that activation fails for the narrow scope and succeeds for the wide one.

Tests of this kind are evidence of a specific character. Each one confirms that for the particular records and policies it constructs the program behaves as the proposition says it must. None of them shows that the proposition holds for all records and all policies, and the reader should not read the passing suite as a proof. The propositions of the chapters are proved in the text, and the tests guard against the code departing from the text. The first draft of `AllOf` passed every test that existed when it was written and was nevertheless inconsistent with the proposition, and it was the attempt to prove the proposition and not any test that revealed the gap. The experience is a fair summary of the relation between the two activities.

9.5 Recording Refusals

Part I ended with a limitation: the ledger could record that an operation was performed and had no honest way to record that one was refused. The `admit` subcommand closes the gap in the following way. It reads a file of records, inscribes and validates them, builds a policy from its flags, admits the records, prints one line per record with its disposition and the first twelve characters of its receipt, and appends a single ledger record whose details are the number of records and the number that received each disposition. A validation failure is recorded too, under a distinct action name, with the stage and the error, so that an attempt to admit an inconsistent set leaves a trace. If activation fails the failure is placed in the details of the ledger record before the error is returned, in keeping with the ordering argued in Chapter 4.

```
$ golab admit -deny bob -scope lab/a -gate REVIEW -project lab/a records
    .jsonl
admitted     r1  7b942872d24a
refused      r2  1a6eaf32f1b9
unscoped     r3  4280dca9df98
```

```
duplicate    r4  441e98d49986
malformed    r5  0777ddd8d006
gated        r6  0c7bb6278674
active in "lab/a": 1 records
(exit status 0)
$ golab admit -project lab/b records.jsonl
admitted     r1  7b942872d24a
admitted     r2  1a6eaf32f1b9
admitted     r3  4280dca9df98
duplicate    r4  441e98d49986
malformed    r5  0777ddd8d006
admitted     r6  0c7bb6278674
golab: "r3" refers to "r1", which is not in scope "lab/b": active record
    refers to an inactive record
(exit status 1)
$ golab audit
ledger holds: 2 records
$ cat golab-ledger.jsonl
{"seq":1,"time":"2026-10-25T17:08:51Z","action":"admit","subject":"
   records.jsonl","detail":{"active":"1","admitted":"1","duplicate
   ":"1","gated":"1","malformed":"1","records":"6","refused":"1","
   unscoped":"1"},"prev":"","digest":"3331396374
   e6bf1715ae6a552acbe5396f2186c724cb76edbf8685d9292655d4"}
{"seq":2,"time":"2026-10-25T17:08:51Z","action":"admit","subject":"
   records.jsonl","detail":{"activation":"\"r3\" refers to \"r1\",
   which is not in scope \"lab/b\": active record refers to an inactive
    record","admitted":"4","duplicate":"1","malformed":"1","records
   ":"6"},"prev":"3331396374
   e6bf1715ae6a552acbe5396f2186c724cb76edbf8685d9292655d4","digest":"
   ff8151dc5ad0678774c0a5d90ba856f73c7f62bcdaa0b0d7994b9d8463fea168"}
```

Listing 9.2: A recorded session of the `admit` command.

In the first command of the transcript the records are judged under three rules at once. The record by the denied author is refused, the one in another scope is unscoped, the repeat is a duplicate, the one without an author is malformed, and the one containing the gated phrase is held for review. Exactly one record is admitted, and it is the only one active in the scope. The second command applies no rules, admits everything that is not malformed or a duplicate, and then projects the second scope, in which the record that remains refers to a record outside it. Activation fails, the command exits with a nonzero status, and the ledger records both runs.

The transcript also shows what the ledger does not yet record. The ledger entry holds counts and not the individual dispositions, so an auditor can see that one record was refused but not which. It holds no description of the policy that was applied, so a run with one set of flags is indistinguishable in the ledger from a run with another that happened to yield the same counts. And the receipts that the command prints are not stored. These omissions are deliberate simplifications for the sake of the chapter and are the subject of the exercises. A ledger that is to support an audit needs to record the policy as well as the outcome, because a disposition is a function of both the record and the rules, and a record of the first without

the second cannot be reproduced.

9.6 Where Part II Ends

The tool now distinguishes four things that the shell blurs and that Part I could only name: a record that has been stored, a record that is structurally sound as part of a set, a record that has been judged under a policy, and a set of records that is coherent on a surface. Each has its own type, each stage has its own function, and the dependence of each on the one before it is visible in the signatures. What the tool lacks is also clear. It runs one stage at a time on one thread, and it does not record the policy it applied. It treats a file of records as the whole world, and it offers no way for several operators to submit records at once. Part III takes up concurrent execution, beginning with the worker pool that verifies many files at the same time and the single goroutine that will own the ledger.

Exercises

Exercise 9.1. Change `Admit` so that a refused record does not register as seen, and show an input on which the dispositions differ from those of the present version. Discuss which behaviour is preferable for an audit.

Exercise 9.2. Extend the ledger record written by `admit` to include the full list of identifiers and dispositions, and measure the size of the record for ten thousand records. Propose a representation in which the ledger holds only a digest of the full list and the list is kept elsewhere, and state what the operator must do to check the digest.

Exercise 9.3. Add a description of the policy to the ledger record, produced by a `Describe` method that each policy implements. Consider whether `PolicyFunc` can implement it, and what that implies about adapting plain functions to richer interfaces.

Exercise 9.4. Implement a stage between admission and projection that releases gated records after a named reviewer approves them. Decide which of the existing types should carry the approval, and whether the receipt of a record should change when it is approved.

Part III

Concurrent Execution

Chapter 10

Goroutines and Channels

10.1 What a Goroutine Is

A goroutine is a function running concurrently with the rest of the program. It is started by placing the keyword `go` before a function call, and the call then proceeds on its own while the caller continues with the next statement. The Go runtime schedules goroutines onto a small number of operating-system threads, so they are cheap enough that a program can hold thousands of them, and a goroutine begins with a stack of only a few kilobytes that grows as needed. The cheapness changes how programs are designed, because the cost that makes a thread-per-task design impractical in many languages is largely absent, and a task that is naturally independent can simply be given a goroutine.

Three properties of goroutines are used throughout Part III and should be stated at the outset. First, all the goroutines of a process share one address space, which means they can read and write the same variables, and the difficulty of concurrent programming is the difficulty of doing so safely. Second, when the `main` function returns, the process exits and every other goroutine is abandoned wherever it happened to be, without any chance to finish; a program that wants a goroutine's work must wait for it. Third, a goroutine has no identity visible to the program and no handle by which it can be stopped from outside, so a goroutine ends only when its function returns, and a program that wants to stop one must arrange for the function to notice and return. The second and third properties explain why the tools of this chapter, the wait group and the channel, are about coordination and not about starting work, which is the easy part.

10.2 Why `golab` Needs Them

Verifying a file means reading its bytes and hashing them. For a handful of small files the work is trivial, and for a research directory holding thousands of large measurements it is the dominant cost of the `verify` command. The work is also independent from file to file: the digest of one file does not depend on any other, so nothing in the problem requires the files to be handled in sequence. The sequential version leaves most of the machine idle. While one file is being read the processor waits for the disk, and while its bytes are being hashed the disk waits for the processor, and several files in flight at once keep both busy.

The number of files in flight needs a bound, and the bound is a design decision and not an optimization detail. Every open file uses a descriptor, which the operating system limits, a disk serves many simultaneous readers worse than a few, and an unbounded program that started a goroutine per file would try to open every file in a large directory at the same moment. The standard remedy is a pool, a fixed number of worker goroutines that take jobs from a shared source until the source is exhausted. The package `pool` implements one for `golab`.

```
// Package pool verifies many files at once with a bounded number of
// goroutines. The result for every input file is always present and is
// returned in input order, whatever order the workers happened to finish in.
package pool

import (
    "context"
    "sync"

    "golab/internal/artifact"
)

// Outcome is what happened to one file. Exactly one of Verified and Err is
// meaningful: a nil Err means Verified holds the digest.
type Outcome struct {
    File     artifact.Discovered
    Verified artifact.Verified
    Err      error
}

// Options configures VerifyAll. The zero value is usable and verifies with
// one worker using artifact.Verify.
type Options struct {
    // Workers is the number of goroutines. Values below one mean one.
    Workers int
    // Verify performs the work for one file. It defaults to artifact.Verify
    // and is a field so that tests can substitute a controllable function.
    Verify func(artifact.Discovered) (artifact.Verified, error)
    // OnResult, if set, is called once per file as soon as its outcome is
    // known. It is called from the worker goroutines, so it must be safe for
    // concurrent use, and calls for different files may overlap and may
    // arrive in any order.
```

```
    OnResult func(Outcome)
}

// VerifyAll verifies every file, using at most opts.Workers goroutines at
// a time, and returns one Outcome per input file in input order.
//
// If ctx is cancelled, files that no worker has started are not verified
// and receive ctx.Err() as their error, so the caller still learns about
// every file. A verification already in progress is not interrupted, since
// artifact.Verify takes no context.
func VerifyAll(ctx context.Context, files []artifact.Discovered, opts Options) []Outcome {
    workers := opts.Workers
    if workers < 1 {
        workers = 1
    }
    verify := opts.Verify
    if verify == nil {
        verify = artifact.Verify
    }

    results := make([]Outcome, len(files))
    jobs := make(chan int)
    var wg sync.WaitGroup

    for w := 0; w < workers; w++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            for i := range jobs {
                // Each index is received by exactly one worker, so each
                // element of results is written by exactly one goroutine.
                out := Outcome{File: files[i]}
                if err := ctx.Err(); err != nil {
                    out.Err = err
                } else {
                    out.Verified, out.Err = verify(files[i])
                }
                results[i] = out
                if opts.OnResult != nil {
                    opts.OnResult(out)
                }
            }
        }()
    }
```

```
77
78      // Feed every index, even after cancellation: the workers record
        the
79      // cancellation as each file's outcome, and no file is silently
        dropped.
80      for i := range files {
81          jobs <- i
82      }
83      close(jobs)
84      wg.Wait()
85      return results
86  }
```

Listing 10.1: A bounded worker pool that verifies files concurrently.

10.3 Reading the Pool

The function `VerifyAll` makes three decisions about structure, each of which is a decision about who may touch what. It allocates the slice of results before any goroutine starts, with one element per input file. It creates an unbuffered channel `jobs` of integers, which will carry indices into the input. And it starts the workers, each a goroutine running the same closure, which receives indices from the channel until it is closed, processes the file at that index, and writes its outcome into the slot of the results slice with the same index.

A channel is a typed conduit through which goroutines pass values. A send on an unbuffered channel blocks until some other goroutine is ready to receive, and a receive blocks until some other goroutine sends, so the pair of operations is both a transfer of a value and a rendezvous between the two goroutines. The loop `for i := range jobs` receives repeatedly and ends when the channel has been closed and drained. Closing is the sender's business, and it is a signal to the receivers that no more values will arrive, so `VerifyAll` closes `jobs` after sending the last index. The rule that only the sender closes a channel is more than a convention, because sending on a closed channel causes a panic, and a receiver that closed a channel would be exposing every sender to it.

A `sync.WaitGroup` counts goroutines still running. The function calls `wg.Add(1)` before starting each worker and each worker calls `wg.Done()` in a deferred call when it returns, and `wg.Wait()` blocks until the count returns to zero. Placing the call to `Add` in the loop that starts the goroutines, and not inside the goroutines themselves, matters: if a goroutine called `Add` on its own behalf, the count could be zero when `Wait` was reached simply because the goroutine had not yet run.

The pool avoids two classical hazards by construction. It does not use a lock to protect the results, because no two goroutines ever touch the same element: each index is sent on the channel once and received by exactly one worker, which is therefore the sole writer of that slot. And it does not capture a loop variable in a goroutine closure. Before Go 1.22 a variable declared in a `for` clause was shared by all iterations, so a goroutine started in the loop that referred to it would see whatever value it held when the goroutine eventually ran. The module declares Go 1.21 and so has the older rule, and the code is written to give the

same result under either rule, by sending the index through a channel and not by capturing it.

10.4 Why the Caller Can Read the Results

Writing to a slice from several goroutines and reading it from another raises the question of what the reader is guaranteed to see. The Go memory model answers it with a small set of ordering rules. A send on a channel happens before the corresponding receive completes, the closing of a channel happens before a receive that returns because the channel is closed, and a call to `Done` on a wait group happens before a `Wait` that it releases returns. An operation that happens before another is guaranteed to be visible to it, and two accesses to the same memory with no such ordering between them, at least one of which is a write, constitute a data race, whose effect the language does not define.

Proposition 10.1. *When `VerifyAll` returns, the result slice has one element per input file, element i is the outcome for file i, and the caller's reads of the slice are ordered after every worker's writes to it.*

Proof. The feeder sends each index i exactly once, so exactly one worker receives it and writes element i, and no other goroutine accesses that element while workers are running. Every worker's last action is its deferred `Done`, and `VerifyAll` returns only after `Wait` has returned, so each write to an element happens before `Done`, which happens before `Wait` returns, which happens before the caller's reads. Because the feeder sends one index per file and receives block until sent to, every element is written before the function returns. □

The proposition explains what would go wrong if the call to `Wait` were removed. The function would return while workers were still writing, and the caller's reads of the slice would have no ordering with those writes, which is a data race even if the program happened to produce the right output on a given run.

10.5 A Race, Observed

The danger is easy to demonstrate and worth seeing once. The program in `examples/ch10_race` starts a thousand goroutines, each of which increments a shared counter, and prints the total, which ought to be one thousand.

```
1  package main
2
3  import (
4      "fmt"
5      "sync"
6  )
7
8  func main() {
9      counter := 0
10     var wg sync.WaitGroup
```

```
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            counter++ // unsynchronized read-modify-write from many goroutines
        }()
    }
    wg.Wait()
    fmt.Println("counter:", counter)
}
```

Listing 10.2: A deliberately racy program.

```
$ go run ./examples/ch10_race
counter: 1000
counter: 998
counter: 995
counter: 988
counter: 983
$ go run -race ./examples/ch10_race
==================
WARNING: DATA RACE
Read at 0x00c000112078 by goroutine 10:
  main.main.func1()
      examples/ch10_race/main.go:15 +0x84

Previous write at 0x00c000112078 by goroutine 7:
  main.main.func1()
      examples/ch10_race/main.go:15 +0x96

Goroutine 10 (running) created at:
  main.main()
      examples/ch10_race/main.go:13 +0x78

[remainder of the report omitted]
exit status 66
```

Listing 10.3: The racy program, run five times and then under the race detector.

The five runs in Listing 10.3 give five different answers, and only the first is correct. The statement `counter++` is three steps, a read, an addition, and a write, and when two goroutines interleave their steps, one of the increments is lost. The result is not wrong in a consistent way that a test could pin down, and a program with this defect can pass a test suite for months. The race detector, enabled with the `-race` flag, instruments every memory access and reports a pair of accesses that are not ordered by the memory model, naming the line, the goroutines involved, and where they were created. A program in which it reports a race exits with status 66.

The detector is a tool and not a proof, and the book relies on it only in that sense. It

reports races that occur in the executions it observes, so a race on a path the tests never exercise is invisible to it, and a program that passes under `-race` has been shown free of races only on the runs that were made. The tests of Part III are therefore designed to drive concurrent paths hard, with many goroutines and many iterations, and are run with the detector enabled, and the arguments in the text, like the one above, are what establish that the code is correct for all runs.

10.6 Goroutines Are Not Services

It is tempting to treat a goroutine as a small independent service, a worker that is handed a job and returns a result, and the vocabulary of the pool encourages it. The comparison conceals the differences that matter most. Goroutines share memory and share fate: a panic in one goroutine that is not recovered terminates the entire process, a runaway goroutine slows every other, and a bug that corrupts a shared structure corrupts it for all of them. A service in the sense used in Part V is a separately deployed program that communicates over a network, and its failure modes are of a different kind. A message can be lost or delivered twice, the remote program can be slow or absent or running an older version, and a request can be processed even though the reply never arrives, so that the sender cannot tell whether the work was done. None of these exists between goroutines, and designs that move from the first to the second without confronting them inherit failures that local concurrency never exposed. The pool of this chapter is a technique for using the processors and disks of one machine, and it makes no claim about distribution.

```
1  package pool
2
3  import (
4      "context"
5      "errors"
6      "fmt"
7      "sync"
8      "sync/atomic"
9      "testing"
10     "time"
11
12     "golab/internal/artifact"
13 )
14
15 func files(n int) []artifact.Discovered {
16     out := make([]artifact.Discovered, n)
17     for i := range out {
18         out[i] = artifact.Discovered{Path: fmt.Sprintf("f%02d", i),
       Kind: artifact.KindSource, Size: int64(i)}
19     }
20     return out
21 }
22
23 func okVerify(d artifact.Discovered) (artifact.Verified, error) {
```

```
    return artifact.Verified{}, nil
}

func TestResultsAreInInputOrderAndComplete(t *testing.T) {
    in := files(100)
    out := VerifyAll(context.Background(), in, Options{Workers: 8, Verify: okVerify})
    if len(out) != len(in) {
        t.Fatalf("got %d outcomes for %d files", len(out), len(in))
    }
    for i, o := range out {
        if o.File.Path != in[i].Path {
            t.Fatalf("outcome %d is for %s, want %s", i, o.File.Path, in[i].Path)
        }
    }
}

func TestWorkersReallyRunInParallel(t *testing.T) {
    const workers = 4
    arrived := make(chan struct{}, workers)
    release := make(chan struct{})
    verify := func(d artifact.Discovered) (artifact.Verified, error) {
        arrived <- struct{}{}
        <-release
        return artifact.Verified{}, nil
    }
    done := make(chan []Outcome)
    go func() { done <- VerifyAll(context.Background(), files(8), Options{Workers: workers, Verify: verify}) }()

    for i := 0; i < workers; i++ {
        select {
        case <-arrived:
        case <-time.After(5 * time.Second):
            t.Fatalf("only %d workers were running at once, want %d", i, workers)
        }
    }
    close(release)
    if got := <-done; len(got) != 8 {
        t.Fatalf("got %d outcomes", len(got))
    }
}

func TestConcurrencyIsBounded(t *testing.T) {
    const workers = 3
    var running, peak atomic.Int32
    verify := func(d artifact.Discovered) (artifact.Verified, error) {
```

```
69          n := running.Add(1)
70          for {
71              p := peak.Load()
72              if n <= p || peak.CompareAndSwap(p, n) {
73                  break
74              }
75          }
76          time.Sleep(2 * time.Millisecond)
77          running.Add(-1)
78          return artifact.Verified{}, nil
79      }
80      VerifyAll(context.Background(), files(60), Options{Workers:
    workers, Verify: verify})
81      if p := peak.Load(); p > workers {
82          t.Fatalf("peak concurrency %d exceeds %d workers", p, workers
    )
83      }
84  }
85
86  func TestCancelledContextStillReportsEveryFile(t *testing.T) {
87      ctx, cancel := context.WithCancel(context.Background())
88      cancel()
89      var calls atomic.Int32
90      verify := func(d artifact.Discovered) (artifact.Verified, error)
    {
91          calls.Add(1)
92          return artifact.Verified{}, nil
93      }
94      out := VerifyAll(ctx, files(20), Options{Workers: 4, Verify:
    verify})
95      if len(out) != 20 {
96          t.Fatalf("got %d outcomes", len(out))
97      }
98      for i, o := range out {
99          if !errors.Is(o.Err, context.Canceled) {
100             t.Fatalf("outcome %d: err = %v, want context.Canceled", i,
     o.Err)
101         }
102     }
103     if calls.Load() != 0 {
104         t.Fatalf("verify was called %d times after cancellation",
    calls.Load())
105     }
106 }
107
108 func TestCancellationMidRun(t *testing.T) {
109     ctx, cancel := context.WithCancel(context.Background())
110     var once sync.Once
111     verify := func(d artifact.Discovered) (artifact.Verified, error)
    {
112         once.Do(cancel) // the first file cancels everything
```

```
        return artifact.Verified{}, nil
    }
    out := VerifyAll(ctx, files(50), Options{Workers: 1, Verify: verify})
    var ok, cancelled int
    for _, o := range out {
        switch {
        case o.Err == nil:
            ok++
        case errors.Is(o.Err, context.Canceled):
            cancelled++
        default:
            t.Fatalf("unexpected error %v", o.Err)
        }
    }
    if ok != 1 || cancelled != 49 {
        t.Fatalf("ok=%d cancelled=%d, want 1 and 49", ok, cancelled)
    }
}

func TestOnResultIsCalledOncePerFile(t *testing.T) {
    var mu sync.Mutex
    seen := map[string]int{}
    opts := Options{Workers: 6, Verify: okVerify, OnResult: func(o Outcome) {
        mu.Lock()
        seen[o.File.Path]++
        mu.Unlock()
    }}
    VerifyAll(context.Background(), files(200), opts)
    if len(seen) != 200 {
        t.Fatalf("callbacks for %d files, want 200", len(seen))
    }
    for p, n := range seen {
        if n != 1 {
            t.Fatalf("%s reported %d times", p, n)
        }
    }
}

func TestVerifiesRealFilesConcurrently(t *testing.T) {
    dir := t.TempDir()
    for i := 0; i < 30; i++ {
        writeFile(t, dir, fmt.Sprintf("n%02d.txt", i), fmt.Sprintf("content %d", i))
    }
    found, err := artifact.Discover(dir)
    if err != nil {
        t.Fatal(err)
    }
    out := VerifyAll(context.Background(), found, Options{Workers:
```

```
    5})
    for _, o := range out {
        if o.Err != nil || o.Verified.IsZero() {
            t.Fatalf("%s: %v", o.File.Path, o.Err)
        }
    }
}
```

Listing 10.4: Tests for the worker pool.

Exercises

Exercise 10.1. Remove the call to `wg.Wait()` from `VerifyAll` and run the tests under `-race`. Report what the detector says and explain it in terms of Proposition 10.1.

Exercise 10.2. The test `TestWorkersReallyRunInParallel` does not measure time. Explain how it proves that the workers run simultaneously, and why a test that compared elapsed times would be less reliable.

Exercise 10.3. Change the pool so that the workers pull file names from a shared slice protected by a mutex instead of receiving indices from a channel. Compare the two designs for clarity and for the guarantees available to the caller.

Exercise 10.4. Write a program that starts a goroutine which panics and show that the process terminates even though `main` is waiting normally. Then add a `recover` in the goroutine and discuss whether recovering is appropriate for a verification worker.

Chapter 11

Contexts, Select, and Cancellation

11.1 The Problem of Stopping

Starting concurrent work is easy and stopping it is hard. A goroutine cannot be killed from outside, as Chapter 10 noted, so any work that may need to be abandoned must be written to notice a request to stop and to return of its own accord. The request has to travel from whoever wants the work stopped to whoever is doing it, and the path may pass through several layers of functions and several goroutines, none of which knows about the others. The standard library solves the problem with a single type, `context.Context`, which carries a cancellation signal and an optional deadline down through a program, and which has become the first parameter of nearly every function in Go that can block.

A context is created from another. The function `context.Background` returns an empty context that is never cancelled and serves as the root. The function `context.WithCancel` derives from a parent a child context together with a function that cancels it, and `context.WithTimeout` derives a child that cancels itself when a duration has elapsed. Cancellation flows downward: cancelling a context cancels every context derived from it, and cancelling a child leaves the parent untouched. The tree of contexts therefore mirrors the tree of work, and cancelling a branch stops the work beneath it and nothing else.

Two methods are used to observe a context. The method `Done` returns a channel that is closed when the context is cancelled, which makes cancellation something a goroutine can wait for with the `select` statement described below. The method `Err` returns `nil` while the context is live and afterwards returns `context.Canceled` or `context.DeadlineExceeded`, saying why it ended, and it is the cheaper form when the code only needs to ask whether to continue. The runner of Chapter 5 already used a context with a timeout to bound a child process, and the pool applies the same idea to a whole batch of files.

11.2 The Select Statement

A goroutine often needs to wait for more than one thing, such as a result, a cancellation, or a timer, and to act on whichever happens first. The `select` statement does this. It lists several

channel operations, blocks until one of them can proceed, and runs the corresponding branch. If several can proceed at the same moment, the runtime picks one at random, so a program must not depend on an order among simultaneously ready cases, and a `default` branch, if present, runs immediately when none is ready, which turns the statement into a non-blocking poll.

The tests of the pool use `select` where a bounded wait is needed. In `TestWorkersReallyRunInParallel`, the test waits for each of four workers to announce that it has arrived at the verification function, and each wait is a `select` between the arrival channel and a timer created by `time.After`. If the workers were serialized, the second arrival would never come, the timer would fire after five seconds, and the test would fail with a message that says how many workers were running. A test that waited on the channel alone would hang forever in that case, and a hung test is a poor way to report a defect.

11.3 The Pool's Cancellation Contract

The pool takes a context as its first argument, in the usual position, and its documentation states what cancellation does. Files that no worker has started when the context ends are not verified, and each of them receives the context's error as its outcome. A verification already under way is not interrupted. Two decisions are contained in that sentence.

The first is that cancellation does not shorten the result. The feeder loop continues to send every index even after the context ends, and the workers, seeing the cancelled context, record the cancellation as the outcome of each remaining file instead of verifying it. The alternative design would stop feeding at the first sign of cancellation and return a short slice, or a slice with empty elements for the files never reached. Either would leave the caller to work out which files had been skipped, and for a tool whose purpose is an honest record, a silent omission is the worst kind of result. The choice is the one made in Chapter 4 for partial failure: every file ends up with an outcome, and the outcome says what happened, which in this case is that it was not attempted because time ran out.

The second decision is that a running verification is not interrupted, because `artifact.Verify` takes no context and reads the file to the end. For ordinary files the delay is small, and for a very large file on a slow disk it is not. The honest statement is that the timeout bounds when new work is started and does not bound when the call returns, and the documentation says so. Making verification itself cancellable is straightforward in principle, since the copy loop can check the context between blocks, and it is the first exercise of this chapter.

```
package pool

import (
    "context"
    "errors"
    "fmt"
    "sync"
    "sync/atomic"
    "testing"
    "time"
```

```
11
12      "golab/internal/artifact"
13  )
14
15  func files(n int) []artifact.Discovered {
16      out := make([]artifact.Discovered, n)
17      for i := range out {
18          out[i] = artifact.Discovered{Path: fmt.Sprintf("f%02d", i),
        Kind: artifact.KindSource, Size: int64(i)}
19      }
20      return out
21  }
22
23  func okVerify(d artifact.Discovered) (artifact.Verified, error) {
24      return artifact.Verified{}, nil
25  }
26
27  func TestResultsAreInInputOrderAndComplete(t *testing.T) {
28      in := files(100)
29      out := VerifyAll(context.Background(), in, Options{Workers: 8,
        Verify: okVerify})
30      if len(out) != len(in) {
31          t.Fatalf("got %d outcomes for %d files", len(out), len(in))
32      }
33      for i, o := range out {
34          if o.File.Path != in[i].Path {
35              t.Fatalf("outcome %d is for %s, want %s", i, o.File.Path,
         in[i].Path)
36          }
37      }
38  }
39
40  func TestWorkersReallyRunInParallel(t *testing.T) {
41      const workers = 4
42      arrived := make(chan struct{}, workers)
43      release := make(chan struct{})
44      verify := func(d artifact.Discovered) (artifact.Verified, error)
        {
45          arrived <- struct{}{}
46          <-release
47          return artifact.Verified{}, nil
48      }
49      done := make(chan []Outcome)
50      go func() { done <- VerifyAll(context.Background(), files(8),
        Options{Workers: workers, Verify: verify}) }()
51
52      for i := 0; i < workers; i++ {
53          select {
54          case <-arrived:
55          case <-time.After(5 * time.Second):
56              t.Fatalf("only %d workers were running at once, want %d",
```

```
        i, workers)
57          }
58      }
59      close(release)
60      if got := <-done; len(got) != 8 {
61          t.Fatalf("got %d outcomes", len(got))
62      }
63  }
64
65  func TestConcurrencyIsBounded(t *testing.T) {
66      const workers = 3
67      var running, peak atomic.Int32
68      verify := func(d artifact.Discovered) (artifact.Verified, error)
        {
69          n := running.Add(1)
70          for {
71              p := peak.Load()
72              if n <= p || peak.CompareAndSwap(p, n) {
73                  break
74              }
75          }
76          time.Sleep(2 * time.Millisecond)
77          running.Add(-1)
78          return artifact.Verified{}, nil
79      }
80      VerifyAll(context.Background(), files(60), Options{Workers:
        workers, Verify: verify})
81      if p := peak.Load(); p > workers {
82          t.Fatalf("peak concurrency %d exceeds %d workers", p, workers
        )
83      }
84  }
85
86  func TestCancelledContextStillReportsEveryFile(t *testing.T) {
87      ctx, cancel := context.WithCancel(context.Background())
88      cancel()
89      var calls atomic.Int32
90      verify := func(d artifact.Discovered) (artifact.Verified, error)
        {
91          calls.Add(1)
92          return artifact.Verified{}, nil
93      }
94      out := VerifyAll(ctx, files(20), Options{Workers: 4, Verify:
        verify})
95      if len(out) != 20 {
96          t.Fatalf("got %d outcomes", len(out))
97      }
98      for i, o := range out {
99          if !errors.Is(o.Err, context.Canceled) {
100             t.Fatalf("outcome %d: err = %v, want context.Canceled", i,
         o.Err)
```

```
        }
    }
    if calls.Load() != 0 {
        t.Fatalf("verify was called %d times after cancellation",
    calls.Load())
    }
}

func TestCancellationMidRun(t *testing.T) {
    ctx, cancel := context.WithCancel(context.Background())
    var once sync.Once
    verify := func(d artifact.Discovered) (artifact.Verified, error)
    {
        once.Do(cancel) // the first file cancels everything
        return artifact.Verified{}, nil
    }
    out := VerifyAll(ctx, files(50), Options{Workers: 1, Verify:
    verify})
    var ok, cancelled int
    for _, o := range out {
        switch {
        case o.Err == nil:
            ok++
        case errors.Is(o.Err, context.Canceled):
            cancelled++
        default:
            t.Fatalf("unexpected error %v", o.Err)
        }
    }
    if ok != 1 || cancelled != 49 {
        t.Fatalf("ok=%d cancelled=%d, want 1 and 49", ok, cancelled)
    }
}

func TestOnResultIsCalledOncePerFile(t *testing.T) {
    var mu sync.Mutex
    seen := map[string]int{}
    opts := Options{Workers: 6, Verify: okVerify, OnResult: func(o
    Outcome) {
        mu.Lock()
        seen[o.File.Path]++
        mu.Unlock()
    }}
    VerifyAll(context.Background(), files(200), opts)
    if len(seen) != 200 {
        t.Fatalf("callbacks for %d files, want 200", len(seen))
    }
    for p, n := range seen {
        if n != 1 {
            t.Fatalf("%s reported %d times", p, n)
        }
```

```
    }
}

func TestVerifiesRealFilesConcurrently(t *testing.T) {
    dir := t.TempDir()
    for i := 0; i < 30; i++ {
        writeFile(t, dir, fmt.Sprintf("n%02d.txt", i), fmt.Sprintf("content %d", i))
    }
    found, err := artifact.Discover(dir)
    if err != nil {
        t.Fatal(err)
    }
    out := VerifyAll(context.Background(), found, Options{Workers: 5})
    for _, o := range out {
        if o.Err != nil || o.Verified.IsZero() {
            t.Fatalf("%s: %v", o.File.Path, o.Err)
        }
    }
}
```

Listing 11.1: Tests for the worker pool, repeated for reference.

Two tests pin down the contract. `TestCancelledContextStillReportsEveryFile` cancels the context before the call and checks that all twenty files come back with `context.Canceled` and that the verification function was never invoked. `TestCancellationMidRun` is more delicate. It uses one worker and a verification function whose first call cancels the context through a `sync.Once`, and it checks that exactly one file was verified and the other forty-nine were cancelled. The test is deterministic only because it uses one worker. With several workers, the number of files already in progress when the cancellation lands varies from run to run, and a test that asserted an exact count would be flaky. The contract promises only that every file has an outcome and that files not started are cancelled, and the tests assert precisely that and no more.

11.4 Cancellation in the Command

The `verify` command exposes the mechanism through the flag `-timeout`. When it is given, the command derives a context with that timeout and passes it to the pool. The flag `-workers` sets the size of the pool, defaulting to the number of processors, and the flag `-each` asks for one ledger record per file, which is the subject of the next chapter.

```
$ golab verify -workers 4 -each papers
c5494ba4ec9f848191fecf8b242f0e8bc98223d4024f1c503aa628f845558766  papers/d01.txt
ee27a25b535e97d09cfa78a1b8ce59a971d777bcc0beaab0848176c04f3be1e5  papers/d02.txt
7e23a4226f0c7a614760d70c37b1340fb5480461da98449d94732e9d59b0a68a  papers
```

```
   /d03.txt
390dd0e18d4337f20285e5117e7c3f055a2cb9405258fcbfd78719d7c80a8d66  papers
   /d04.txt
ba1ea46e22c396aabf3c078a773755fb1f61d047dfaac6ba7a1188ad1cc9f144  papers
   /d05.txt
6a0da86687d0fb29f564a2ab9aad047cd1d3eec42cd00dcd6dae84ed50223c22  papers
   /d06.txt
$ golab audit
ledger holds: 7 records
$ cut -d, -f1,3,4 golab-ledger.jsonl
{"seq":1,"action":"verify-file","subject":"papers/d01.txt"
{"seq":2,"action":"verify-file","subject":"papers/d04.txt"
{"seq":3,"action":"verify-file","subject":"papers/d03.txt"
{"seq":4,"action":"verify-file","subject":"papers/d02.txt"
{"seq":5,"action":"verify-file","subject":"papers/d05.txt"
{"seq":6,"action":"verify-file","subject":"papers/d06.txt"
{"seq":7,"action":"verify","subject":"papers"
$ golab verify -workers 4 -timeout 1ns papers
golab: 6 files could not be verified: context deadline exceeded; context
    deadline exceeded; context deadline exceeded; context deadline
   exceeded; context deadline exceeded; context deadline exceeded
(exit status 1)
$ tail -n 1 golab-ledger.jsonl | cut -d, -f3-6
"action":"verify","subject":"papers","detail":{"failed":"6","found":"6"
```

Listing 11.2: Concurrent verification, per-file records, and a timeout that expires at once.

The transcript in Listing 11.2 shows all three effects. The first command verifies six files with four workers and writes a record for each file followed by the aggregate record, seven in all, and the audit confirms that the chain holds. The order in which the per-file records appear is not the order of the files: `d04` precedes `d03` and `d02`, because the ledger records files in the order the workers finished them, and the order of completion is a property of the scheduler that the program does not control. The printed digests, by contrast, are in input order, because the command prints from the result slice after the pool has returned. The two orders serve different readers, and the difference is intended. The ledger is a history of what happened, and the terminal output is a report in a stable order.

The second command sets a timeout of one nanosecond, which expires before any work can start. All six files come back with the deadline error, the command exits with status 1, and the aggregate record in the ledger states that six files failed. The attempt is on the record. An operator who looks at the ledger a week later sees that a verification was requested, that nothing was verified, and why, which is the same principle that governed the ordering of the `verify` command in Chapter 4.

Exercises

Exercise 11.1. Change `artifact.Verify` to accept a context and to check it between blocks of a fixed size while hashing. Choose the block size and justify it, and write a test that cancels a verification of a large file part-way through. Decide what the returned error

should say about the bytes already read.

Exercise 11.2. Write a version of the feeder that stops sending as soon as the context is cancelled and fills the unsent slots with the context's error afterwards. Show that it satisfies the same contract, and identify which design is easier to reason about.

Exercise 11.3. Replace the use of `ctx.Err()` in the worker with a `select` on `ctx.Done()` and a `default` case. Explain whether the two behave differently, and why one might prefer either.

Exercise 11.4. Make the workers' error for a cancelled file wrap `context.Canceled` or `context.DeadlineExceeded` with the path of the file, and show that `errors.Is` still recognizes the cause.

Chapter 12

Shared State and the Single Writer

12.1 The Fork That Part I Predicted

Chapter 6 ended with a warning. The ledger's `Append` reads the whole file, checks the chain, computes the next record, and writes it, and these steps are not one atomic action. Two processes that run them at about the same time can each read the same last record and each write a successor, so that the chain acquires two records with the same sequence number. The warning can be tested by launching twenty copies of the command at once, each of which runs a trivial program and records the outcome.

```
$ for i in $(seq 20); do golab run papers true & done; wait
$ wc -l < golab-ledger.jsonl
5
$ golab audit
golab: record 5 has seq 4: ledger chain broken
(exit status 1)
$ cut -c1-9 golab-ledger.jsonl
{"seq":1,
{"seq":2,
{"seq":3,
{"seq":4,
{"seq":4,
```

Listing 12.1: Twenty processes appending to one ledger.

The result in Listing 12.1 is the failure predicted. Of twenty commands, five managed to append, and two of them wrote sequence number four. The audit reports the damage at the fifth record. The remaining fifteen commands exited with an error, because by the time they ran, the ledger already failed its check and `Append` refuses to extend a broken chain. The behaviour is not an accident of the transcript. The exact numbers depend on timing and would differ on another run, but the shape is general: once two writers collide, the ledger is damaged, the damage is detected, and every later writer is turned away. The design failed closed, and the cost was an unusable ledger.

The defect is a race of the kind the previous chapters studied, with the difference that the shared state is a file and the competing parties are processes and not goroutines. The pool

of Chapter 10 avoided shared writes by giving each slot of the result slice to one goroutine. The ledger cannot be partitioned that way, because every record depends on the one before it, and the whole point of the structure is a single total order. A total order needs exactly one place at which the order is decided.

12.2 Two Ways to Serialize

There are two standard ways to make a shared resource safe for many goroutines. The first is a lock: a `sync.Mutex` guards the resource, every user acquires it before touching the resource and releases it afterward, and the lock ensures that only one goroutine is inside the guarded section at a time. The second is ownership: a single goroutine is given exclusive responsibility for the resource, and every other goroutine asks it to act by sending a message over a channel. Go programmers summarize the second style with a proverb, that one should share memory by communicating and not communicate by sharing memory. The proverb is a preference and not a rule, and either style can be right. A lock is the simpler tool for protecting a small piece of state with a few operations. Ownership suits a resource that has a lifecycle, an order of operations, and a single point where work should be sequenced, and the ledger has all three.

12.3 The Writer

The type `ledger.Writer` implements ownership. Its constructor starts one goroutine, the loop, which receives requests from a channel and calls the existing `Append` function for each, one at a time, so that within the process no two appends ever overlap. Every other goroutine calls the method `Append` on the writer, which packages the arguments into a request, sends it, and waits for a reply. The listing also contains a method `Snapshot`, which sends a request of a second kind that asks the loop to read the whole ledger instead of appending to it. It is not needed in this chapter and is explained in Chapter 16, where a service has readers that run while appends are in progress.

```
1  package ledger
2
3  import (
4      "errors"
5      "sync"
6      "time"
7  )
8
9  // ErrClosed is returned by Append on a Writer that has been closed.
10 var ErrClosed = errors.New("ledger writer closed")
11
12 // Writer owns a ledger file for the lifetime of a process. Every
       append is
13 // sent over a channel to one goroutine, which is the only code in
       the
```

```
// process that touches the file. Many goroutines may call Append at once;
// the writer serializes them, so the chain of digests cannot fork inside
// this process.
//
// A Writer protects nothing against other processes. Two processes that
// each hold a Writer on the same file can still fork the chain.
type Writer struct {
    path string
    reqs chan request
    done chan struct{}

    mu     sync.RWMutex // guards closed, and orders sends on reqs against Close
    closed bool
}

type request struct {
    snapshot bool // a read of the whole ledger, not an append
    action   string
    subject  string
    detail   map[string]string
    reply    chan result
}

type result struct {
    rec  Record
    recs []Record
    err  error
}

// NewWriter starts the goroutine that owns the file at path. The buffer
// size is the number of requests that may queue before Append callers wait.
func NewWriter(path string, buffer int) *Writer {
    w := &Writer{
        path: path,
        reqs: make(chan request, buffer),
        done: make(chan struct{}),
    }
    go w.loop()
    return w
}

func (w *Writer) loop() {
    defer close(w.done)
    for req := range w.reqs {
        if req.snapshot {
```

```
            recs, err := Load(w.path)
            req.reply <- result{recs: recs, err: err}
            continue
        }
        rec, err := Append(w.path, req.action, req.subject, req.
    detail, time.Now())
        req.reply <- result{rec: rec, err: err} // reply is buffered,
     so this never blocks
    }
}

// Append writes one record and returns it once it is on disk. It may
     be
// called from any number of goroutines. The detail map is copied
    before it
// is queued, so the caller may reuse it.
func (w *Writer) Append(action, subject string, detail map[string]
    string) (Record, error) {
    req := request{
        action:  action,
        subject: subject,
        detail:  copyDetail(detail),
        reply:   make(chan result, 1),
    }

    // Holding the read lock while sending guarantees that Close
    cannot
    // close the channel under a sender. Close takes the write lock,
    so it
    // waits for sends already in progress, and the loop keeps
    receiving, so
    // those sends do complete.
    w.mu.RLock()
    if w.closed {
        w.mu.RUnlock()
        return Record{}, ErrClosed
    }
    w.reqs <- req
    w.mu.RUnlock()

    res := <-req.reply
    return res.rec, res.err
}

// Snapshot returns every record in the ledger as of a moment between
     two
// appends. The read is performed by the owning goroutine, so it
    never sees a
// record that is only partly written. It does not check the chain.
func (w *Writer) Snapshot() ([]Record, error) {
    req := request{snapshot: true, reply: make(chan result, 1)}
```

```
    w.mu.RLock()
    if w.closed {
        w.mu.RUnlock()
        return nil, ErrClosed
    }
    w.reqs <- req
    w.mu.RUnlock()
    res := <-req.reply
    return res.recs, res.err
}

// Close stops accepting appends, writes every request already accepted,
// and waits for the owning goroutine to finish. It is safe to call more
// than once.
func (w *Writer) Close() error {
    w.mu.Lock()
    if !w.closed {
        w.closed = true
        close(w.reqs)
    }
    w.mu.Unlock()
    <-w.done
    return nil
}

func copyDetail(m map[string]string) map[string]string {
    if m == nil {
        return nil
    }
    out := make(map[string]string, len(m))
    for k, v := range m {
        out[k] = v
    }
    return out
}
```

Listing 12.2: The single-writer ledger.

Several details of Listing 12.2 carry weight. Each request contains its own reply channel with a buffer of one, so the loop can deliver the result without waiting for the caller to be ready, and a caller that has gone away cannot block the loop or leave a goroutine stranded. The detail map is copied before it is queued, for the reason given in Chapter 7: a map is a reference, and a caller that reused its map for the next call would otherwise change a request the writer had not yet processed. The timestamp is taken inside the loop, at the moment of writing, and not by the caller, so the times in the ledger never run backward relative to the sequence numbers.

The subtle part is shutdown. A channel can be closed to tell its receivers that no more requests will come, and closing it ends the loop, but sending on a closed channel panics,

so closing the channel while another goroutine is about to send would crash the program. The writer prevents this with a read-write lock. Each call to `Append` holds the read lock while it checks the closed flag and sends its request, and `Close` takes the write lock to set the flag and close the channel. The write lock waits until all current holders of the read lock have released it, which means all in-flight sends have completed, and the loop is still receiving during that time, so they do complete. Once `Close` holds the lock and has closed the channel, any later `Append` sees the flag and returns `ErrClosed` without sending. The loop drains whatever is already in the channel before it ends, so every request accepted before the close is written.

Proposition 12.1. *Suppose a single `Writer` is the only code appending to its file. Then every call to `Writer.Append` that returns a nil error corresponds to exactly one record on disk, the records appear in the order in which the loop received the requests, and the chain of digests remains valid. After `Close` returns, no further record is written.*

Proof. The loop is the only goroutine that calls `ledger.Append` on the file, and it handles one request completely before receiving the next, so appends are sequential and each reads the file as left by the one before. Each is the same function whose correctness for sequential use was established by the tests of Chapter 6. A request is replied to after its append returns, so success reported to a caller means the record was written. The channel is closed only under the write lock, after which the closed flag prevents any send, and the loop ends only after the channel is drained, so requests accepted before the close are processed and requests after it are refused. □

The proposition has a hypothesis, and the hypothesis is the limit of the design. The writer protects the file from goroutines in one process. It does nothing about a second process, which has no knowledge of the first writer's goroutine and will read and write the file on its own account. Twenty copies of the command, each with its own writer, would reproduce the failure of Listing 12.1 exactly. Protecting the file against other processes needs a mechanism from the operating system, such as an advisory lock on the file, or an arrangement in which only one process ever writes, and the second of those is the shape that Part V gives the tool by making a service the sole owner of the ledger and having clients send it requests over a network. The first exercise explores the lock.

```
1  package ledger
2
3  import (
4      "errors"
5      "fmt"
6      "path/filepath"
7      "sync"
8      "testing"
9  )
10
11 func TestWriterSerializesConcurrentAppends(t *testing.T) {
12     path := filepath.Join(t.TempDir(), "ledger.jsonl")
13     w := NewWriter(path, 8)
14
```

```
    const goroutines, each = 20, 10
    var wg sync.WaitGroup
    for g := 0; g < goroutines; g++ {
        wg.Add(1)
        go func(g int) {
            defer wg.Done()
            for i := 0; i < each; i++ {
                _, err := w.Append("event", fmt.Sprintf("g%d-%d", g, i), map[string]string{"g": fmt.Sprint(g)})
                if err != nil {
                    t.Errorf("append: %v", err)
                    return
                }
            }
        }(g)
    }
    wg.Wait()
    if err := w.Close(); err != nil {
        t.Fatal(err)
    }

    records, err := Load(path)
    if err != nil {
        t.Fatal(err)
    }
    if len(records) != goroutines*each {
        t.Fatalf("%d records, want %d", len(records), goroutines*each)
    }
    if err := Check(records); err != nil {
        t.Fatalf("chain broken by concurrent writers: %v", err)
    }
    subjects := map[string]bool{}
    for _, r := range records {
        subjects[r.Subject] = true
    }
    if len(subjects) != goroutines*each {
        t.Fatalf("%d distinct subjects, want %d: records were lost or repeated", len(subjects), goroutines*each)
    }
}

func TestAppendAfterCloseFails(t *testing.T) {
    w := NewWriter(filepath.Join(t.TempDir(), "l.jsonl"), 1)
    if err := w.Close(); err != nil {
        t.Fatal(err)
    }
    if _, err := w.Append("x", "y", nil); !errors.Is(err, ErrClosed) {
        t.Fatalf("want ErrClosed, got %v", err)
    }
```

```
    if err := w.Close(); err != nil {
        t.Fatalf("second Close: %v", err)
    }
}

func TestCloseWhileAppendersAreActive(t *testing.T) {
    path := filepath.Join(t.TempDir(), "ledger.jsonl")
    w := NewWriter(path, 2)

    var wg sync.WaitGroup
    var accepted, refused sync.Map
    for g := 0; g < 16; g++ {
        wg.Add(1)
        go func(g int) {
            defer wg.Done()
            for i := 0; i < 20; i++ {
                subject := fmt.Sprintf("g%d-%d", g, i)
                if _, err := w.Append("event", subject, nil); err == nil {
                    accepted.Store(subject, true)
                } else if errors.Is(err, ErrClosed) {
                    refused.Store(subject, true)
                } else {
                    t.Errorf("unexpected error: %v", err)
                }
            }
        }(g)
    }
    go w.Close() // closes at an arbitrary moment during the appends
    wg.Wait()
    w.Close()

    records, err := Load(path)
    if err != nil {
        t.Fatal(err)
    }
    if err := Check(records); err != nil {
        t.Fatal(err)
    }
    // Every append that reported success is on disk, and nothing else is.
    n := 0
    accepted.Range(func(_, _ any) bool { n++; return true })
    if n != len(records) {
        t.Fatalf("%d appends reported success but the ledger holds %d records", n, len(records))
    }
}

func TestSnapshotSeesCompletedAppendsAndNeverATornRecord(t *testing.T) {
```

```
    path := filepath.Join(t.TempDir(), "l.jsonl")
    w := NewWriter(path, 8)
    defer w.Close()

    var wg sync.WaitGroup
    for g := 0; g < 8; g++ {
        wg.Add(1)
        go func(g int) {
            defer wg.Done()
            for i := 0; i < 25; i++ {
                if _, err := w.Append("a", fmt.Sprintf("g%d-%d", g, i), nil); err != nil {
                    t.Error(err)
                    return
                }
            }
        }(g)
    }
    last := 0
    for i := 0; i < 50; i++ {
        recs, err := w.Snapshot()
        if err != nil {
            t.Fatalf("snapshot during appends: %v", err)
        }
        if err := Check(recs); err != nil {
            t.Fatal(err)
        }
        if len(recs) < last {
            t.Fatalf("snapshot shrank from %d to %d", last, len(recs))
        }
        last = len(recs)
    }
    wg.Wait()
    recs, _ := w.Snapshot()
    if len(recs) != 200 {
        t.Fatalf("len = %d, want 200", len(recs))
    }
    w.Close()
    if _, err := w.Snapshot(); !errors.Is(err, ErrClosed) {
        t.Fatalf("snapshot after close: %v", err)
    }
}
```

Listing 12.3: Tests for the single-writer ledger.

12.4 What the Tests Establish

The first test starts twenty goroutines that each append ten records through one writer, then checks that all two hundred records are present, that the chain holds, and that every subject appears exactly once, so that no record was lost and none was written twice. The test was run with the race detector enabled. The third test is the one that exercises the shutdown protocol. It starts sixteen goroutines appending twenty records each, calls `Close` from another goroutine at an arbitrary moment, and checks that the number of appends that reported success equals the number of records on disk and that the chain is valid. Every append is either accepted and written or refused with `ErrClosed`, and none is lost in between. As with every concurrency test in the book, the test samples some interleavings and does not enumerate them, and the argument that the protocol is correct is the one in Proposition 12.1.

With a writer in place the command can offer what Chapter 4 said it lacked. The flag `-each` gives the `OnResult` callback of the pool a function that appends a record for each file as soon as the file's outcome is known. The callbacks run in the worker goroutines, so up to as many appends may be requested at once as there are workers, and the writer serializes them. The transcript in Chapter 11 shows the resulting records, in order of completion and with a valid chain, followed by the aggregate record.

12.5 Costs and Where Part III Ends

The writer is correct and not fast. Each call to the underlying `Append` reads and checks the entire ledger, so the cost of the nth append is proportional to n and the cost of n appends is proportional to n^2. For a ledger of a few thousand records the delay is imperceptible, and for a ledger of a million it is prohibitive. A writer that holds the head digest and the next sequence number in memory, and checks the whole chain only when it opens the file, would make each append constant in cost, at the price of trusting its own memory between appends. The trade is a common one in systems that keep an authoritative log, and the third exercise asks for it.

Part III has given the tool concurrency in three forms, a bounded pool for independent work, a context for cancellation, and a single writer for the one resource that must be sequenced. Each is accompanied by a statement of what it does not do. The pool does not interrupt a running verification, the context bounds the start of work and not its end, and the writer does not protect against other processes. Part IV turns to data that must outlive a process, and the question there will be which of what `golab` knows should be stored, and which can always be rebuilt.

Exercises

Exercise 12.1. Add an advisory file lock around the body of `ledger.Append`, using the facility your operating system provides, and rerun the twenty-process experiment. Report what changes, and identify one situation in which the lock fails to protect the chain, such as a ledger on a network filesystem.

Exercise 12.2. Replace the single-writer goroutine with a `sync.Mutex` around `ledger.Append`, and compare the two designs with respect to shutdown, to the order of records, and to the amount of code. State which you would choose and why.

Exercise 12.3. Make the writer keep the head digest and the next sequence number in memory, checking the chain on disk only when the writer is created. Show that appending a thousand records is faster, and describe a scenario in which the in-memory state and the file disagree.

Exercise 12.4. Remove the read lock from `Writer.Append` and write a test that calls `Close` while appenders are active, run under the race detector and repeated many times. Describe the failure that appears and relate it to the rule about closing channels.

Part IV

Memory and Reconstruction

Chapter 13

Encoding, Buffering, and Atomic Replacement

13.1 Data That Outlives a Process

Everything built so far either finishes within one run of the program or goes into the ledger, which is written once and never rewritten. Part IV is concerned with a third kind of data, which is written repeatedly, replaced as a whole, and read by later runs of the program or by other programs. A file of this kind raises a set of questions that the ledger avoided by being append-only. How is a structure turned into bytes and back without losing or inventing anything? What happens when the bytes are large? What does a reader see if it opens the file while a writer is part-way through replacing it, and what is left behind if the machine stops in the middle? Go answers the first question with the `encoding` packages, the second with `bufio`, and the third with no package at all, only a technique that depends on a property of the operating system, which this chapter states exactly and then tests as far as a test can.

13.2 Encoding as a Round Trip

The ledger already uses `encoding/json`, and it is worth saying what the package does with a struct, because the later chapters depend on it. The encoder writes the exported fields of a struct in the order of their declaration, uses the name in the field tag when there is one, and omits a field marked `omitempty` when its value is the zero value. It writes the keys of a map in sorted order. These rules make the encoding of a value a function of the value alone, which is why the digests of Chapter 6 are reproducible. The decoder is more permissive than the encoder. It ignores fields it does not recognize, leaves absent fields at their zero values, and accepts the fields in any order, so a file written by a newer version of a program is read without complaint by an older one, and the fields the older version does not know about are silently dropped when it writes the file again.

That permissiveness is convenient and also a hazard, and the index of the next chapter guards against it with an explicit version number and with checks on the decoded value, for instance that no path appears twice. The general point is that a round trip through an

encoding preserves exactly what the struct can express. Information that the struct has no field for does not survive, and the compiler has nothing to say about it. A reader who wants a round trip to be faithful has two honest options, which are to decode into a type that rejects unknown fields by calling `DisallowUnknownFields` on the decoder, or to state in the documentation that unknown fields are discarded. The index takes the second course, because an index is derived data and a field it fails to preserve can be recomputed.

13.3 Buffering and a Limit That Was There All Along

Reading and writing a file one small piece at a time is slow, because each call to the operating system has a fixed cost. The package `bufio` wraps a reader or writer in a buffer so that many small operations become few large ones. A `bufio.Writer` holds its data until the buffer fills or until `Flush` is called, and the last of these is the one that is easy to forget: bytes written to a buffered writer and not flushed are lost without any error. A `bufio.Scanner` reads a file in lines, tokens, or words, and the ledger uses it to read one JSON record per line.

The scanner has a limit that the ledger of Part I did not mention. A scanner holds a line in a buffer of bounded size, and when a line is longer than the limit it stops with the error `bufio.ErrTooLong`. The ledger sets the limit at one mebibyte, which is far more than any record the program produced in the earlier parts. The limit is nonetheless a property of the reader that the writer did not respect. `Append` would write a record of any length, and a record longer than the limit could therefore be written successfully and then never read back, and since `Append` begins by loading the whole ledger, every later append would fail as well. A single oversized detail value, such as a long error message, would disable the ledger permanently.

The repair is small and the lesson is general. The ledger package now declares the limit as an exported constant, `MaxLine`, sets the scanner’s buffer from it, and refuses in `Append` to write a record whose encoding exceeds it, returning the sentinel `ErrTooLarge`. The new test appends a record that is too large, checks that the error is the sentinel, and then checks that a normal append still succeeds and that the ledger loads. The principle is that a writer must not produce what its own reader cannot consume. Where the format has a limit, the limit belongs in one place, and the code that writes must check it as well as the code that reads.

13.4 Replacing a File Whole

The ledger can be appended to because each record is independent of the file’s other contents. A file that is rewritten as a whole, such as the index, is in a different position. The obvious way to rewrite it is to open the file for writing, which truncates it, and write the new contents. That sequence has a window in which the file is empty or partly written. A reader that opens the file in that window sees a fragment, and a crash in that window leaves a fragment behind permanently, with the old contents already destroyed.

The standard remedy is to write the new contents to a different file and then rename it over the old one. On POSIX filesystems, renaming a file to a name that already exists, within

one filesystem, replaces the destination in a single step, so that any process that opens the name sees either the entire old file or the entire new one. The package `atomicfile` packages the technique.

```
// Package atomicfile replaces a file's contents so that a reader, or a
// crash, never leaves the file half written. The new contents are written to
// a temporary file in the same directory, flushed to stable storage, and
// then renamed over the destination.
//
// The guarantee rests on the rename. On POSIX filesystems, renaming within
// one filesystem replaces the destination in a single step, so a reader opens
// either the whole old file or the whole new one. The package does not
// promise anything on filesystems that implement rename differently, and it
// does not make the contents correct; it only makes them arrive whole.
package atomicfile

import (
    "bufio"
    "fmt"
    "io"
    "os"
    "path/filepath"
)

// Write calls fill with a buffered writer for a new file and, if fill and
// every later step succeed, installs the file at path with the given
// permissions. If anything fails, the file at path is left exactly as it was
// and the temporary file is removed.
func Write(path string, perm os.FileMode, fill func(io.Writer) error) (err error) {
    dir, base := filepath.Split(path)
    if dir == "" {
        dir = "."
    }
    tmp, err := os.CreateTemp(dir, base+".tmp-*")
    if err != nil {
        return fmt.Errorf("create temporary file: %w", err)
    }
    tmpName := tmp.Name()
    closed := false
```

```
    defer func() {
        if err != nil {
            if !closed {
                tmp.Close()
            }
            os.Remove(tmpName)
        }
    }()

    w := bufio.NewWriter(tmp)
    if err = fill(w); err != nil {
        return err
    }
    if err = w.Flush(); err != nil {
        return fmt.Errorf("flush: %w", err)
    }
    if err = tmp.Chmod(perm); err != nil {
        return fmt.Errorf("chmod: %w", err)
    }
    if err = tmp.Sync(); err != nil {
        return fmt.Errorf("sync file: %w", err)
    }
    closed = true
    if err = tmp.Close(); err != nil {
        return fmt.Errorf("close: %w", err)
    }
    if err = os.Rename(tmpName, path); err != nil {
        return fmt.Errorf("rename: %w", err)
    }
    // The rename itself is a change to the directory, and it is durable only
    // once the directory has been flushed. A failure here means the new file
    // is in place but might not survive a crash, so it is reported.
    d, derr := os.Open(dir)
    if derr != nil {
        return fmt.Errorf("open directory: %w", derr)
    }
    defer d.Close()
    if err = d.Sync(); err != nil {
        return fmt.Errorf("sync directory: %w", err)
    }
    return nil
}

// WriteBytes is Write for contents already in memory.
func WriteBytes(path string, perm os.FileMode, data []byte) error {
    return Write(path, perm, func(w io.Writer) error {
        _, err := w.Write(data)
        return err
    })
```

```
85 }
```

Listing 13.1: Replacing a file by writing a temporary and renaming it.

The function creates the temporary file in the same directory as the destination, because a rename between directories on different filesystems is a copy and has no such guarantee. It hands the caller a buffered writer through a callback, so that the caller can stream a large structure without first building all its bytes in memory, and it flushes the buffer itself, so that the caller cannot forget. It sets the permissions, calls `Sync` on the file to ask the operating system to write the data to stable storage before the rename, closes the file and checks the error, renames, and finally syncs the directory, because the rename is itself a change to the directory and is durable only once the directory has been flushed.

The cleanup deserves a comment because it uses a feature of the language that is easy to misread. The function has a named result, `err`, and a deferred function that inspects it. Every failing step assigns to that variable before returning, so the deferred function can tell whether the function is failing and, if so, close the temporary file when it has not been closed and remove it. A deferred function runs after the return value has been set, and with a named result it can read what was set, which is how a single piece of cleanup code covers every early return. The variable `closed` records that the file was already closed, since closing a file twice reports an error that would mask the real one.

Proposition 13.1. *Suppose `Write` returns a non-nil error. Then the file at the destination path has the same contents it had before the call, and no temporary file created by the call remains, except possibly when the error comes from the final synchronization of the directory, in which case the new contents are in place.*

Proof. The destination is modified only by the call to `os.Rename`, which comes after every step that can fail except the directory synchronization. If any earlier step fails, the destination has not been touched, and the deferred function removes the temporary file. If the rename itself fails, the destination is unchanged because a rename either takes effect or does not, and the deferred function removes the temporary file. If only the directory synchronization fails, the rename has taken effect, the temporary name no longer exists, and the error reports that the replacement may not survive a crash. □

The proposition is about the function’s behaviour on errors it can observe. It does not cover a crash, since a program that has stopped cannot clean up. A crash before the rename leaves a stray temporary file with a distinctive name and the destination untouched. A crash after the rename but before the directory flush may, on some filesystems, leave the old file in place after the machine restarts. Neither case produces a partial destination file, which is the property the technique exists to provide, and neither can be demonstrated by an ordinary test, because a test that is running has not crashed.

13.5 What the Tests Establish

```
1 package atomicfile
2
```

```
import (
    "bytes"
    "errors"
    "io"
    "os"
    "path/filepath"
    "sync"
    "testing"
)

func names(t *testing.T, dir string) []string {
    t.Helper()
    es, err := os.ReadDir(dir)
    if err != nil {
        t.Fatal(err)
    }
    var out []string
    for _, e := range es {
        out = append(out, e.Name())
    }
    return out
}

func TestWriteCreatesAndReplaces(t *testing.T) {
    dir := t.TempDir()
    p := filepath.Join(dir, "x.json")
    if err := WriteBytes(p, 0o640, []byte("one")); err != nil {
        t.Fatal(err)
    }
    if err := WriteBytes(p, 0o640, []byte("two")); err != nil {
        t.Fatal(err)
    }
    got, _ := os.ReadFile(p)
    if string(got) != "two" {
        t.Fatalf("got %q", got)
    }
    st, _ := os.Stat(p)
    if st.Mode().Perm() != 0o640 {
        t.Fatalf("mode %v", st.Mode().Perm())
    }
    if n := names(t, dir); len(n) != 1 {
        t.Fatalf("leftover files: %v", n)
    }
}

func TestFailedFillLeavesOldFileAndNoTemporary(t *testing.T) {
    dir := t.TempDir()
    p := filepath.Join(dir, "x.json")
    if err := WriteBytes(p, 0o644, []byte("old contents")); err !=
   nil {
        t.Fatal(err)
```

```
    }
    boom := errors.New("boom")
    err := Write(p, 0o644, func(w io.Writer) error {
        w.Write(bytes.Repeat([]byte("partial"), 100000)) // more than the buffer holds
        return boom
    })
    if !errors.Is(err, boom) {
        t.Fatalf("err = %v, want boom", err)
    }
    got, _ := os.ReadFile(p)
    if string(got) != "old contents" {
        t.Fatalf("old file damaged: %d bytes", len(got))
    }
    if n := names(t, dir); len(n) != 1 {
        t.Fatalf("leftover files: %v", n)
    }
}

func TestMissingDirectoryIsAnError(t *testing.T) {
    p := filepath.Join(t.TempDir(), "nope", "x")
    if err := WriteBytes(p, 0o644, []byte("a")); err == nil {
        t.Fatal("expected an error")
    }
}

// A reader that runs while the file is being replaced must see one whole
// version or the other, never a mixture or a prefix.
func TestReadersNeverSeeAPartialFile(t *testing.T) {
    dir := t.TempDir()
    p := filepath.Join(dir, "x")
    a := bytes.Repeat([]byte("a"), 1<<20)
    b := bytes.Repeat([]byte("b"), 1<<20)
    if err := WriteBytes(p, 0o644, a); err != nil {
        t.Fatal(err)
    }

    stop := make(chan struct{})
    var wg sync.WaitGroup
    bad := make(chan string, 1)
    wg.Add(1)
    go func() {
        defer wg.Done()
        for {
            select {
            case <-stop:
                return
            default:
            }
            got, err := os.ReadFile(p)
```

```
            if err != nil {
                select {
                case bad <- err.Error():
                default:
                }
                return
            }
            if !bytes.Equal(got, a) && !bytes.Equal(got, b) {
                select {
                case bad <- "mixed or truncated contents":
                default:
                }
                return
            }
        }
    }()
    for i := 0; i < 40; i++ {
        data := b
        if i%2 == 1 {
            data = a
        }
        if err := WriteBytes(p, 0o644, data); err != nil {
            t.Fatal(err)
        }
    }
    close(stop)
    wg.Wait()
    select {
    case msg := <-bad:
        t.Fatal(msg)
    default:
    }
}
```

Listing 13.2: Tests for atomic replacement.

The tests cover what an ordinary test can reach. The first checks that a file is created and then replaced, that the permissions are applied, and that no extra files remain in the directory. The second makes the callback write more bytes than the buffer holds, so that some of them reach the temporary file, and then return an error. It checks that the old contents are intact and that nothing is left over, which tests the first sentence of the proposition at the point where it is most at risk. The third checks that a missing directory is reported as an error. The fourth runs a reader in a loop while the main goroutine replaces a one-mebibyte file forty times, alternating between two contents that consist entirely of one repeated letter, and checks that every read returns one of the two complete contents. The test samples the interleavings of one run on one filesystem and does not prove the property for every filesystem, and the reader of the chapter should treat the rename guarantee as one that the operating system makes and that the test merely fails to contradict.

Exercises

Exercise 13.1. Change the ledger's `Load` to use a `json.Decoder` reading directly from the file, and not a scanner, so that no line limit exists. Decide whether the constant `MaxLine` and the error `ErrTooLarge` should then be removed, and justify the decision in terms of what a very large record would do to the memory use of `Append`.

Exercise 13.2. Remove the call to `Sync` on the temporary file and describe a failure that the tests of this chapter cannot detect. Explain why the order of the synchronization and the rename matters.

Exercise 13.3. Make the temporary file in a different directory from the destination, on a different filesystem if one is available, and report what the rename does. State what this implies about the correct location of the temporary file.

Exercise 13.4. Modify the decoding of the index so that unknown fields are an error, using `DisallowUnknownFields`. Describe a situation in which this makes an upgrade of the program harder, and a situation in which it catches a real mistake.

Chapter 14

Derived Data and the Index

14.1 Authority and Convenience

The ledger answers questions slowly. To learn the last digest recorded for a given file, a program must read every record from the start and keep the last one that mentions the file. That is acceptable for a ledger of a few hundred records and tiresome for a ledger of a million. The usual response is to keep a second structure, an index, that holds the answer to the common questions directly. The response is safe or unsafe according to a single decision, which is whether the program treats the index as a source of truth or as a summary of one. In the first case the index must be protected, backed up, and kept consistent with everything else, and a loss of the index is a loss of information. In the second case the index is derived data. It may be deleted at any time, and the only cost is the time to compute it again.

The index of `golab` is of the second kind, and the design is arranged to make that status checkable and not merely declared. Three rules are enforced in the code. The index is computed from the ledger records alone and from nothing else. It records which prefix of the ledger it was computed from, by the number of records and the digest of the last one. And the program never makes a decision from the index that it could not make by recomputing the index from the ledger, a rule that holds because every path that uses the index first obtains it from a function that falls back on rebuilding.

14.2 The Package

```
// Package index derives a lookup table from the ledger. The table is a
// convenience and never an authority: it is computed entirely from the ledger
// records, it carries the digest of the last record it has seen so that its
// age can be checked, and it can be deleted at any time and rebuilt to
// exactly the same bytes.
package index
```

```
7
8  import (
9      "encoding/json"
10     "errors"
11     "fmt"
12     "os"
13     "sort"
14
15     "golab/internal/atomicfile"
16     "golab/internal/ledger"
17 )
18
19 // Version is written into every index file. A file with another
       version is
20 // not trusted and is rebuilt.
21 const Version = 1
22
23 // Errors reported when an index cannot be used as it stands. None of
        them is
24 // fatal to a caller that can fall back on Build.
25 var (
26     ErrOutOfSequence = errors.New("record does not follow the index")
27     ErrStale         = errors.New("index does not belong to this
       ledger")
28 )
29
30 // Entry summarizes everything the ledger says about one subject path
       .
31 type Entry struct {
32     Path string `json:"path"`
33     // LastDigest is the digest from the most recent successful
       verification
34     // of the path, whether or not a later attempt failed.
35     LastDigest string `json:"last_digest,omitempty"`
36     // Verified is true when the most recent attempt succeeded.
37     Verified bool `json:"verified"`
38     Attempts int  `json:"attempts"`
39     Failures int  `json:"failures"`
40     // Seq is the sequence number of the last record that touched the
        path.
41     Seq int `json:"seq"`
42 }
43
44 // Index is the derived table. Records and Head identify the prefix
       of the
45 // ledger it was computed from.
46 type Index struct {
47     records int
48     head    string
49     byPath  map[string]*Entry
50 }
```

```

type wire struct {
    Version int      `json:"version"`
    Records int      `json:"records"`
    Head    string   `json:"head"`
    Entries []Entry `json:"entries"`
}

// New returns the index of an empty ledger.
func New() *Index { return &Index{byPath: map[string]*Entry{}} }

// Records is the number of ledger records the index has consumed.
func (x *Index) Records() int { return x.records }

// Head is the digest of the last record consumed, empty for none.
func (x *Index) Head() string { return x.head }

// Len is the number of distinct paths in the index.
func (x *Index) Len() int { return len(x.byPath) }

// Lookup returns a copy of the entry for path.
func (x *Index) Lookup(path string) (Entry, bool) {
    e, ok := x.byPath[path]
    if !ok {
        return Entry{}, false
    }
    return *e, true
}

// Entries returns copies of all entries sorted by path.
func (x *Index) Entries() []Entry {
    out := make([]Entry, 0, len(x.byPath))
    for _, e := range x.byPath {
        out = append(out, *e)
    }
    sort.Slice(out, func(i, j int) bool { return out[i].Path < out[j].
    Path })
    return out
}

// Apply consumes the next ledger record. The record must be the
    immediate
// successor of the last one consumed, which is how the index
    guarantees that
// it describes a contiguous prefix of one chain.
func (x *Index) Apply(r ledger.Record) error {
    if r.Seq != x.records+1 || r.Prev != x.head {
        return fmt.Errorf("record %d: %w", r.Seq, ErrOutOfSequence)
    }
    switch r.Action {
    case "verify-file", "verify-file-failed":
```

```
        e := x.byPath[r.Subject]
        if e == nil {
            e = &Entry{Path: r.Subject}
            x.byPath[r.Subject] = e
        }
        e.Attempts++
        e.Seq = r.Seq
        if r.Action == "verify-file" {
            e.Verified = true
            e.LastDigest = r.Detail["sha256"]
        } else {
            e.Verified = false
            e.Failures++
        }
    }
    x.records = r.Seq
    x.head = r.Digest
    return nil
}

// Build computes the index of a whole ledger. It refuses a ledger whose
// chain does not hold, because a table derived from a damaged record would
// look authoritative and be nothing of the kind.
func Build(records []ledger.Record) (*Index, error) {
    if err := ledger.Check(records); err != nil {
        return nil, err
    }
    x := New()
    for _, r := range records {
        if err := x.Apply(r); err != nil {
            return nil, err
        }
    }
    return x, nil
}

// CatchUp brings an index forward to the end of records. It first checks that
// the index belongs to this ledger, meaning that the record at its position
// has the digest it remembers, and returns ErrStale if not. The remaining
// records are checked as the chain is extended.
func (x *Index) CatchUp(records []ledger.Record) error {
    if x.records > len(records) {
        return fmt.Errorf("index has seen %d records, ledger holds %d: %w", x.records, len(records), ErrStale)
    }
    if x.records > 0 && records[x.records-1].Digest != x.head {
```

```
144         return fmt.Errorf("record %d has a different digest: %w", x.
    records, ErrStale)
145     }
146     if err := ledger.Check(records); err != nil {
147         return err
148     }
149     for _, r := range records[x.records:] {
150         if err := x.Apply(r); err != nil {
151             return err
152         }
153     }
154     return nil
155 }
156 
157 // MarshalJSON writes the index with entries sorted by path, so equal
     indexes
158 // always produce equal bytes.
159 func (x *Index) MarshalJSON() ([]byte, error) {
160     return json.Marshal(wire{Version: Version, Records: x.records,
    Head: x.head, Entries: x.Entries()})
161 }
162 
163 // UnmarshalJSON reads an index and rejects files it does not
    understand.
164 func (x *Index) UnmarshalJSON(b []byte) error {
165     var w wire
166     if err := json.Unmarshal(b, &w); err != nil {
167         return err
168     }
169     if w.Version != Version {
170         return fmt.Errorf("index version %d, want %d", w.Version,
    Version)
171     }
172     m := make(map[string]*Entry, len(w.Entries))
173     for i := range w.Entries {
174         e := w.Entries[i]
175         if _, dup := m[e.Path]; dup {
176             return fmt.Errorf("duplicate entry for %q", e.Path)
177         }
178         m[e.Path] = &e
179     }
180     x.records, x.head, x.byPath = w.Records, w.Head, m
181     return nil
182 }
183 
184 // Save writes the index to path atomically.
185 func (x *Index) Save(path string) error {
186     b, err := json.MarshalIndent(x, "", " ")
187     if err != nil {
188         return err
189     }
```

```
    return atomicfile.WriteBytes(path, 0o644, append(b, '\n'))
}

// Source says how Open obtained an index.
type Source string

const (
    Loaded   Source = "loaded"
    CaughtUp Source = "caught up"
    Rebuilt  Source = "rebuilt"
)

// Open returns an index consistent with records, using the file at path when
// it is usable and rebuilding from the ledger when it is missing, damaged, of
// another version, or not a prefix of this ledger. The returned string gives
// the reason when the index was rebuilt. Open does not write the file.
func Open(path string, records []ledger.Record) (*Index, Source, string, error) {
    b, err := os.ReadFile(path)
    reason := ""
    if err != nil {
        reason = "no index file"
        if !errors.Is(err, os.ErrNotExist) {
            reason = err.Error()
        }
    } else {
        x := New()
        if uerr := json.Unmarshal(b, x); uerr != nil {
            reason = "unreadable index: " + uerr.Error()
        } else {
            behind := x.records < len(records)
            if cerr := x.CatchUp(records); cerr == nil {
                if behind {
                    return x, CaughtUp, "", nil
                }
                return x, Loaded, "", nil
            } else if errors.Is(cerr, ledger.ErrBroken) {
                return nil, "", "", cerr
            } else {
                reason = cerr.Error()
            }
        }
    }
    x, err := Build(records)
    if err != nil {
        return nil, "", "", err
    }
```

```
236     return x, Rebuilt, reason, nil
237 }
```

Listing 14.1: The index, derived from the ledger.

An `Entry` summarizes what the ledger says about one path: the digest from its most recent successful verification, whether the latest attempt succeeded, the number of attempts and failures, and the sequence number of the last record that mentioned it. The fields are chosen for the questions the command will ask, and the entry deliberately keeps the last known digest even when a later attempt failed, so that a path whose verification failed once is not mistaken for a path that was never verified. The `Index` itself holds the entries in a map from path to entry, together with the number of records consumed and the digest of the last. The map is unexported, so that the only ways to change an index are the methods of the package, and every change goes through `Apply`.

The method `Apply` consumes one ledger record and begins by checking that the record is the immediate successor of the last one consumed: its sequence number must be one more than the count, and its predecessor field must equal the remembered head. A record that skips ahead, repeats, or belongs to a different chain is refused with `ErrOutOfSequence`. This check is what makes the pair of numbers a faithful description of the index. The count and the head are not labels attached from outside; they are advanced by the same method that updates the entries, and the method will not advance them over a gap. Only the two verification actions affect entries. Every other record, such as a run or an admission, advances the position and leaves the entries alone, so the index covers the whole ledger even though it summarizes only part of what is in it.

The function `Build` computes the index of a complete ledger by applying every record to an empty index, and it begins by calling the ledger’s `Check`. A table derived from a broken chain would look authoritative while resting on nothing, and the function therefore refuses to produce one. The method `CatchUp` brings an existing index forward. It verifies that the index belongs to the ledger it is given, by checking that the record at the index’s position has the digest the index remembers, then checks the chain and applies the records that follow. A shorter ledger, or a ledger whose record at that position has a different digest, produces `ErrStale`.

The function `Open` ties the pieces together. It reads the index file if there is one, decodes it, and tries to catch it up. If the file is missing, unreadable, of another version, inconsistent, or not a prefix of this ledger, it builds the index from the ledger instead, and it returns the reason. The result says which of the three things happened, so that a command can tell the operator whether the index was loaded as it stood, caught up with new records, or rebuilt. The one case in which `Open` does not fall back is a ledger that fails its own check, because there is nothing authoritative left to rebuild from, and the error is returned as it is.

14.3 Reconstruction Equals Incremental Update

An index that can be updated by appending records and also rebuilt from scratch raises an obvious question, which is whether the two routes give the same result. If they could differ, the index would depend on its history, and the claim that it is derived from the ledger alone

would be false.

Proposition 14.1. *Let L be a ledger of n records that satisfies `Check`, and let $0 \le k \le n$. The encoding of the index obtained by building from the first k records and then calling `CatchUp` with L is byte for byte the encoding of the index built from all of L.*

Proof. `Build` is the fold of `Apply` over the records from the empty index, and `CatchUp` continues the same fold from the position k over the remaining records. The state of an index is its count, its head, and its map of entries, and `Apply` computes the new state from the old state and the record by a deterministic rule that consults nothing else, no clock, no file, and no iteration over the map. After n applications the two routes have applied the same rule to the same records in the same order starting from equal states, so they end in equal states. The encoding of an index depends only on its state, because the entries are written sorted by path and the count and head are written as they stand. Equal states therefore have equal encodings. □

The proof rests on facts that the code could lose without the compiler noticing. If `Apply` read the current time, or consulted an environment variable, or built part of its state by ranging over a map and so depended on the randomized iteration order, the proof would break and the proposition would be false. The sorted output of `Entries` exists for the same reason, and the test that checks the proposition does so for every cut point k from zero to the length of the ledger. As in the earlier chapters, the test checks instances and the argument covers the cases, and the argument is only as good as the discipline that keeps `Apply` pure.

```
package index

import (
    "bytes"
    "encoding/json"
    "errors"
    "fmt"
    "os"
    "path/filepath"
    "testing"
    "time"

    "golab/internal/ledger"
)

// chain builds a valid ledger of the given actions without touching
   disk.
func chain(t *testing.T, specs ...[3]string) []ledger.Record {
    t.Helper()
    path := filepath.Join(t.TempDir(), "l.jsonl")
    now := time.Unix(1_700_000_000, 0)
    for _, s := range specs {
        var d map[string]string
        if s[0] == "verify-file" {
```

```
            d = map[string]string{"sha256": s[2]}
        }
        if _, err := ledger.Append(path, s[0], s[1], d, now); err != nil {
            t.Fatal(err)
        }
    }
    rs, err := ledger.Load(path)
    if err != nil {
        t.Fatal(err)
    }
    return rs
}

func enc(t *testing.T, x *Index) []byte {
    t.Helper()
    b, err := json.Marshal(x)
    if err != nil {
        t.Fatal(err)
    }
    return b
}

var sample = [][3]string{
    {"verify-file", "a.txt", "d1"},
    {"verify-file", "b.txt", "d2"},
    {"run", "echo hi", ""},
    {"verify-file-failed", "b.txt", ""},
    {"verify-file", "a.txt", "d3"},
    {"verify", "dir", ""},
}

func TestBuildSummarizesPerPath(t *testing.T) {
    x, err := Build(chain(t, sample...))
    if err != nil {
        t.Fatal(err)
    }
    a, _ := x.Lookup("a.txt")
    if a.LastDigest != "d3" || !a.Verified || a.Attempts != 2 || a.Seq != 5 {
        t.Fatalf("a = %+v", a)
    }
    b, _ := x.Lookup("b.txt")
    if b.LastDigest != "d2" || b.Verified || b.Failures != 1 || b.Seq != 4 {
        t.Fatalf("b = %+v", b)
    }
    if x.Records() != 6 || x.Len() != 2 {
        t.Fatalf("records=%d len=%d", x.Records(), x.Len())
    }
}
```

```
72
73 // Folding the records one at a time, from any prefix, gives the same
       index as
74 // building from all of them at once.
75 func TestIncrementalEqualsRebuild(t *testing.T) {
76     rs := chain(t, sample...)
77     want := enc(t, mustBuild(t, rs))
78     for cut := 0; cut <= len(rs); cut++ {
79         t.Run(fmt.Sprint(cut), func(t *testing.T) {
80             x := mustBuild(t, rs[:cut])
81             if err := x.CatchUp(rs); err != nil {
82                 t.Fatal(err)
83             }
84             if got := enc(t, x); !bytes.Equal(got, want) {
85                 t.Fatalf("cut %d differs:\n%s\n%s", cut, got, want)
86             }
87         })
88     }
89 }
90
91 func mustBuild(t *testing.T, rs []ledger.Record) *Index {
92     t.Helper()
93     x, err := Build(rs)
94     if err != nil {
95         t.Fatal(err)
96     }
97     return x
98 }
99
100 func TestSaveLoadRoundTripIsByteIdentical(t *testing.T) {
101     rs := chain(t, sample...)
102     p := filepath.Join(t.TempDir(), "idx.json")
103     if err := mustBuild(t, rs).Save(p); err != nil {
104         t.Fatal(err)
105     }
106     first, _ := os.ReadFile(p)
107
108     x, src, _, err := Open(p, rs)
109     if err != nil || src != Loaded {
110         t.Fatalf("src=%v err=%v", src, err)
111     }
112     if err := x.Save(p); err != nil {
113         t.Fatal(err)
114     }
115     second, _ := os.ReadFile(p)
116     if !bytes.Equal(first, second) {
117         t.Fatal("round trip changed the bytes")
118     }
119
120     // Deleting the file and rebuilding gives the same bytes again.
121     os.Remove(p)
```

```
    y, src, reason, err := Open(p, rs)
    if err != nil || src != Rebuilt || reason == "" {
        t.Fatalf("src=%v reason=%q err=%v", src, reason, err)
    }
    y.Save(p)
    third, _ := os.ReadFile(p)
    if !bytes.Equal(first, third) {
        t.Fatal("rebuilt index differs from the original")
    }
}

func TestApplyRejectsGapsAndForeignRecords(t *testing.T) {
    rs := chain(t, sample...)
    x := New()
    if err := x.Apply(rs[1]); !errors.Is(err, ErrOutOfSequence) {
        t.Fatalf("skipping a record: %v", err)
    }
    x.Apply(rs[0])
    other := chain(t, [3]string{"run", "other", ""}, [3]string{"run",
    "other2", ""})
    if err := x.Apply(other[1]); !errors.Is(err, ErrOutOfSequence) {
        t.Fatalf("record from another ledger: %v", err)
    }
}

func TestOpenCatchesUpAndDetectsForeignLedger(t *testing.T) {
    rs := chain(t, sample...)
    p := filepath.Join(t.TempDir(), "idx.json")
    mustBuild(t, rs[:3]).Save(p)

    x, src, _, err := Open(p, rs)
    if err != nil || src != CaughtUp || x.Records() != len(rs) {
        t.Fatalf("src=%v n=%d err=%v", src, x.Records(), err)
    }

    // An index saved from a different ledger of the same length is
    stale.
    other := chain(t, [3]string{"run", "x", ""}, [3]string{"run", "y",
    ""}, [3]string{"run", "z", ""},
        [3]string{"run", "w", ""}, [3]string{"run", "v", ""}, [3]
    string{"run", "u", ""})
    mustBuild(t, other).Save(p)
    _, src, reason, err := Open(p, rs)
    if err != nil || src != Rebuilt || reason == "" {
        t.Fatalf("src=%v reason=%q err=%v", src, reason, err)
    }
}

func TestOpenRebuildsFromDamagedOrOldFiles(t *testing.T) {
    rs := chain(t, sample...)
    dir := t.TempDir()
```

```
    for name, content := range map[string]string{
        "garbage":  "not json",
        "version":  `{"version":99,"records":0,"head":"","entries":[]}`,
        "dupe":     `{"version":1,"records":0,"head":"","entries":[{"path":"a"},{"path":"a"}]}`,
        "too-long": `{"version":1,"records":50,"head":"zz","entries":[]}`,
    } {
        p := filepath.Join(dir, name)
        os.WriteFile(p, []byte(content), 0o644)
        _, src, reason, err := Open(p, rs)
        if err != nil || src != Rebuilt || reason == "" {
            t.Errorf("%s: src=%v reason=%q err=%v", name, src, reason, err)
        }
    }
}

func TestBuildRefusesABrokenLedger(t *testing.T) {
    rs := chain(t, sample...)
    rs[2].Subject = "tampered"
    if _, err := Build(rs); !errors.Is(err, ledger.ErrBroken) {
        t.Fatalf("err = %v", err)
    }
    p := filepath.Join(t.TempDir(), "idx.json")
    mustBuild(t, chain(t, sample...)).Save(p)
    if _, _, _, err := Open(p, rs); !errors.Is(err, ledger.ErrBroken) {
        t.Fatalf("Open on a broken ledger: %v", err)
    }
}
```

Listing 14.2: Tests for the index.

14.4 What the Index Does Not Protect Against

The chain of digests gives the index a way to notice that it belongs to a different ledger or has fallen behind, and it gives no way to notice that someone edited the index. The head and the count are checked against the ledger. The entries are not, because the check that would compare them is the check that would rebuild them. An index file whose head and count match the ledger but whose entries have been altered is accepted by `Open` and used. The fault is not in the code, which does what it says. It follows from the structure: a summary that is cheaper than the source cannot be verified against the source without paying the cost of the source.

Two responses are available, and the book takes the first. The index is treated as untrusted input whose effect is bounded. The command that uses it, which the next chapter presents, always hashes the files on disk itself and uses the index only to remember what

the ledger last recorded, so a falsified entry can produce a wrong report that a file changed or did not change, and it cannot produce a record in the ledger or a decision to admit anything. The command that writes the index offers a flag that rebuilds it unconditionally, which is the remedy when an operator has reason to doubt the file. The second response is to store in the index a digest of its own entries, which detects accidental damage and not a deliberate edit by someone who can recompute it, and the third exercise asks for it.

Exercises

Exercise 14.1. Extend `Entry` with the time of the first and last attempts, taken from the ledger records. Explain why the time must come from the record and not from the clock at the moment of indexing, and show how the proof of Proposition 14.1 would fail if it came from the clock.

Exercise 14.2. Add a second index, from action name to count, and show that the single pass in `Build` can fill both. Discuss whether the two should be one type or two, and how `Open` would decide that either is stale.

Exercise 14.3. Store in the index file a digest of its entries and check it on load. State exactly which edits the check detects and which it does not, and explain why it does not make the index authoritative.

Exercise 14.4. The method `CatchUp` calls `Check` on the whole ledger on every call, which makes catching up proportional to the size of the ledger and not to the number of new records. Propose a variant that checks only the new records and the link to the old head, state what it assumes about the old records, and say when that assumption is unsafe.

Chapter 15

Reconstruction and Its Limits

15.1 Two Commands

Two commands put the index to use. The command `index` obtains an index that agrees with the ledger, by loading, catching up, or rebuilding as `Open` decides, writes it to disk with the atomic replacement of Chapter 13, and reports which of the three happened and why. The command `status` answers the question that motivated the index, which is whether the files in a directory still match what the ledger last recorded about them. It hashes every file under the directory with the pool of Part III, looks each path up in the index, and prints one line per path with a state. A path is *unchanged* when its current digest equals the last recorded digest and the latest attempt succeeded. It is *changed* when the digests differ, *new* when the ledger has never mentioned it, *missing* when the ledger mentions it under the directory and no file is there now, and *unreadable* when it could not be hashed. Two further states, *never-verified* and *last-attempt-failed*, report paths whose history in the ledger contains only failures or ends in one. The command exits with a failure status when any path is not unchanged, so that a script can use it as a check, and it writes nothing at all: not to the ledger, not to the index.

15.2 A Recorded Session

The session below uses a directory of three small files. The first command verifies them with one worker and with the `-each` flag, so that the ledger receives a record for each file in a deterministic order. The rest of the session builds the index, destroys it, builds it again, and then disturbs the files and the ledger in turn.

```
$ golab verify -workers 1 -each papers
b6a98d9ce9a2d9149288fa3df42d377c3e42737afdcdaf714e33c0a100b51060  papers
   /a.txt
f2c82decdd7181cf98945929a62598db7e6b477e11f6e0eb0ae97020eff151ad  papers
   /b.txt
ae9a6306a205417afddd14316cc1d0d5e04a98f1be10865dce643925ee070ce2  papers
   /c.txt
$ golab index
```

```
index rebuilt (no index file): 3 paths from 4 ledger records
$ head -n 12 golab-index.json
{
 "version": 1,
 "records": 4,
 "head": "
   d945b292fd07e9032a0d91b1fb999e8bb42b2d16122834091bc2ceb621c9b515",
 "entries": [
  {
   "path": "papers/a.txt",
   "last_digest": "
   b6a98d9ce9a2d9149288fa3df42d377c3e42737afdcdaf714e33c0a100b51060",
   "verified": true,
   "attempts": 1,
   "failures": 0,
   "seq": 1
$ sha256sum golab-index.json
b8fa1e53b74a19e32c3d84018a109cf628550db598a082b04f43615143e7f9b8  golab-
   index.json
$ rm golab-index.json && golab index
index rebuilt (no index file): 3 paths from 4 ledger records
$ sha256sum golab-index.json
b8fa1e53b74a19e32c3d84018a109cf628550db598a082b04f43615143e7f9b8  golab-
   index.json
$ golab status papers
unchanged           papers/a.txt
unchanged           papers/b.txt
unchanged           papers/c.txt
$ echo changed > papers/b.txt; echo fresh > papers/d.txt; rm papers/c.
   txt
$ golab status papers
unchanged           papers/a.txt
changed             papers/b.txt
missing             papers/c.txt
new                 papers/d.txt
golab: 3 of 4 paths differ from the ledger
(exit status 1)
$ golab verify -workers 1 -each papers > /dev/null; golab index
index caught up: 4 paths from 8 ledger records
$ golab status papers
unchanged           papers/a.txt
unchanged           papers/b.txt
missing             papers/c.txt
unchanged           papers/d.txt
golab: 1 of 4 paths differ from the ledger
(exit status 1)
$ sed -i '2s/verify-file/verify-fil_/' golab-ledger.jsonl
$ golab status papers
golab: record 2 digest mismatch: ledger chain broken
(exit status 1)
$ golab index
```

```
golab: record 2 digest mismatch: ledger chain broken
(exit status 1)
```

Listing 15.1: Building, destroying, and rebuilding the index, and the states reported by `status`.

The transcript carries the argument of Part IV in a few observations. The first build says that it rebuilt the index because no index file existed, and that it covers three paths from four ledger records, the three file records and the aggregate. The listing of the start of the file shows the version, the count of records, the head digest, and the first entry, in an encoding whose layout is fixed by the declaration of the types. The checksum of the file is printed, the file is deleted, the command is run again, and the checksum is the same. That is Proposition 14.1 observed on one ledger: the file was a function of the ledger, and a function recomputes to the same value.

The session then changes the world. The file `b.txt` is overwritten, a new file `d.txt` is created, and `c.txt` is removed, and `status` reports a changed, a new, and a missing path, with the other still unchanged and an exit status of 1. The report says nothing about why the content of `b.txt` changed or whether the change was legitimate. It says that the bytes on disk no longer match the digest the ledger holds, which is a statement about a difference and not about its cause.

After the files are verified again, the ledger gains four more records, and the command `index` reports that it caught up, with four paths from eight records, so that the existing file was extended and not rebuilt. A new `status` shows `b.txt` and `d.txt` as unchanged and `c.txt` still as missing, and the last of these is correct and worth noticing. The ledger is a history, and it records that `c.txt` was once verified. Deleting the file does not delete the fact that it was verified, and the index, being derived from the ledger, keeps reporting the path until someone records otherwise. A tool that wanted removals to clear the entry would need a ledger action for removal, which is a design decision about the ledger and not a detail of the index.

The last two commands follow the alteration of one letter in the second line of the ledger. Both `status` and `index` stop with the same error, naming record 2 and the broken chain, and neither produces output from the damaged ledger. The index file that existed on disk was not consulted in a way that could have hidden the damage, because `Open` checks the chain before it trusts anything.

15.3 What Was Not Shown

Two properties of the design were checked separately and are reported here without a transcript. The output of `status` is the same whether or not the index file exists. In a directory prepared like the one above, running the command with the file present, deleting the file, and running it again produced identical output, which is what the rule that no decision depends on the index predicts. The second property is the safety of concurrent writers. Thirty simultaneous runs of `index` against one ledger left a single index file with the same checksum as a single run, and no temporary files behind. The result is the one the rename argument gives. Each process writes its own temporary file and renames it over the

destination, every one of the files is a correct index of the same ledger, and so whichever rename lands last, the destination is correct. The contrast with the ledger of Chapter 12 is instructive. The ledger is a sequence, each record depends on the one before, and two writers disagree about what comes next. The index is a function of the ledger, any two writers compute the same value, and a collision between them costs nothing. Derived data is cheap to make safe for concurrency because it has no history of its own to protect.

15.4 Limits of the Analogy

The part is named for memory and reconstruction, and the name needs a statement of what it claims. The engineering discipline is plain: keep one authoritative record, derive everything else from it, and be able to delete the derived forms without loss. The word *reconstruction* is used for that discipline and for nothing more. Nothing in this chapter suggests that an index resembles a memory in how it is formed, how it fails, or what it is for. A human memory is not a function of a log, and it changes when it is used. A cache is exactly a function of a log and does not change. The two share one property, that each is rebuilt from something other than itself, and the book makes no further use of the comparison.

The engineering limits are several. Rebuilding costs time proportional to the ledger, so an index is a saving only when it is loaded and extended more often than it is rebuilt. The derived data is only as good as the authoritative record, and the ledger records what the program was told and what it did, not what is true of the files in the world, as the persistent *missing* entry shows. The index is trusted to the degree that its file has not been edited, which Chapter 14 identified as an unchecked risk with a bounded effect. And the chain beneath everything is tamper-evident only relative to an external copy of its head, a limit that Chapter 6 stated and that nothing in Part IV removes.

Part IV leaves the tool with a durable ledger, a derived index, and a safe way to replace a file. All of it runs inside one machine and is operated by one person at a command line. Part V removes that restriction by making the tool a service that other programs can reach over a network, and the first thing the service will need is what the ledger could not have for itself in Part III: a single owner for the one resource that must be sequenced.

Exercises

Exercise 15.1. Add a ledger action that records the removal of a path, have `status` report a removal as a distinct state, and update `Apply` so that an entry is cleared by a removal record. Decide whether the index should drop the entry or keep it marked as removed, and justify the choice in terms of the proposition on reconstruction.

Exercise 15.2. Write a test that deletes the index file between two runs of the `status` logic, in process, and checks that the lines produced are identical. Then write a deliberately wrong version of `Open` that trusts a stale index, and show that the test fails.

Exercise 15.3. Run thirty copies of `golab index` at once on a ledger that is simultaneously receiving new records from `verify`. Report whether any index file ever fails `Open`, and

explain why the answer does not contradict the single-ledger fork of Chapter 12.

Exercise 15.4. Measure the time taken by `index -rebuild` for ledgers of one thousand, ten thousand, and one hundred thousand records, and plot the results. State at what size the saving from loading an existing index exceeds the cost of writing it.

Part V

Distributed Systems

Chapter 16

HTTP Servers and Handlers

16.1 From a Command to a Service

Through Part IV the tool was something a person ran, which started, did its work, recorded the result, and exited. A service is the same logic with a different lifetime and a different set of callers. It starts once, waits, and handles requests from other programs for as long as it is left running. Two things follow from that change and both reach back to earlier parts of the book. The first is that requests arrive concurrently, so every shared resource needs the discipline of Part III. The second is that the callers are not the author, so everything that arrives is input from outside, and the checks that were conveniences in a command become the boundary of the system.

The standard library's `net/http` package supplies the server, and its design repays attention because it is built from the same ideas as the rest of the book. The central abstraction is the interface `http.Handler`, which has a single method, `ServeHTTP`, taking a response writer and a request. A handler is anything that can answer a request. The package also defines `http.HandlerFunc`, a function type with a `ServeHTTP` method that calls the function, so that any function of the right shape can be used as a handler. That is exactly the adapter pattern of `PolicyFunc` in Chapter 7, and the standard library uses it for the same reason. The service of this chapter is a single value, a `Server`, that implements `http.Handler`, and its routing table is a list of `HandlerFunc` values built from its own methods.

```
1 // Package service exposes golab over HTTP. The structure keeps three
2 // questions apart. Routing chooses a handler from the method and
     path alone.
3 // Adjudication decides whether the caller may use that handler.
     Admission,
4 // inside the handler for records, decides what becomes of the
     records
5 // submitted. Only the last is a judgment about content.
6 package service
7
8 import (
9     "encoding/json"
```

```
    "errors"
    "fmt"
    "io"
    "log/slog"
    "net/http"
    "strconv"
    "time"

    "golab/internal/index"
    "golab/internal/ledger"
    "golab/internal/pipeline"
)

// MaxBody is the largest request body the service reads.
const MaxBody = 1 << 20

// Server holds what the handlers share. The ledger is reached only through
// the Writer, which makes the service the single owner of the ledger within
// its process.
type Server struct {
    Ledger *ledger.Writer
    Policy pipeline.Policy
    Tokens *Tokens
    // Logger receives one structured entry per request. It never receives a
    // credential. A nil Logger discards everything.
    Logger *slog.Logger
    router *Router
}

// New builds the server and its routing table.
func New(w *ledger.Writer, p pipeline.Policy, t *Tokens) *Server {
    s := &Server{Ledger: w, Policy: p, Tokens: t, Logger: slog.New(slog.NewTextHandler(io.Discard, nil))}
    s.router = NewRouter([]Route{
        {http.MethodGet, "/healthz", NeedNothing, http.HandlerFunc(s.health)},
        {http.MethodGet, "/v1/ledger/head", NeedRead, http.HandlerFunc(s.head)},
        {http.MethodGet, "/v1/ledger", NeedRead, http.HandlerFunc(s.list)},
        {http.MethodGet, "/v1/index", NeedRead, http.HandlerFunc(s.lookup)},
        {http.MethodPost, "/v1/admit", NeedWrite, http.HandlerFunc(s.admit)},
    })
    return s
}

```

```
// statusRecorder remembers the status and size of a response so that the
// request can be logged after the handler has finished.
type statusRecorder struct {
    http.ResponseWriter
    status int
    bytes  int
}

func (r *statusRecorder) WriteHeader(code int) {
    r.status = code
    r.ResponseWriter.WriteHeader(code)
}

func (r *statusRecorder) Write(b []byte) (int, error) {
    n, err := r.ResponseWriter.Write(b)
    r.bytes += n
    return n, err
}

// Unwrap lets http.ResponseController reach the underlying writer.
func (r *statusRecorder) Unwrap() http.ResponseWriter { return r.ResponseWriter }

// ServeHTTP routes, then adjudicates, then runs the handler, and logs the
// outcome of every request including the refused ones. Routing comes first,
// so a request for a path that does not exist receives 404 whether or not it
// carries a credential.
func (s *Server) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    rec := &statusRecorder{ResponseWriter: w, status: http.StatusOK}
    start := time.Now()
    principal := ""
    defer func() {
        level := slog.LevelInfo
        switch {
        case rec.status >= 500:
            level = slog.LevelError
        case rec.status >= 400:
            level = slog.LevelWarn
        }
        s.Logger.Log(r.Context(), level, "request",
            "method", r.Method,
            "path", r.URL.Path,
            "status", rec.status,
            "bytes", rec.bytes,
            "duration_ms", float64(time.Since(start).Microseconds())/1000,
            "principal", principal,
```

```
        )
    }()
    w = rec

    route, status, allow := s.router.Match(r.Method, r.URL.Path)
    if route == nil {
        if allow != nil {
            w.Header().Set("Allow", allowHeader(allow))
        }
        writeError(w, status, http.StatusText(status))
        return
    }
    p, authenticated := s.Tokens.Authenticate(r)
    if authenticated {
        principal = p.Name
    }
    if status, ok := Permit(p, authenticated, route.Need); !ok {
        if status == http.StatusUnauthorized {
            w.Header().Set("WWW-Authenticate", `Bearer realm="golab"`
    )
        }
        writeError(w, status, http.StatusText(status))
        return
    }
    route.Handler.ServeHTTP(w, r.WithContext(withPrincipal(r.Context
    (), p)))
}

func writeJSON(w http.ResponseWriter, status int, v any) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(status)
    json.NewEncoder(w).Encode(v) // the status is already sent;
    nothing useful can be done with a failure here
}

func writeError(w http.ResponseWriter, status int, msg string) {
    writeJSON(w, status, map[string]string{"error": msg})
}

// internal answers a failure of the server's own machinery. The
    client is
// told only that something went wrong; the cause, which may contain
    file
// names and system messages, goes to the log.
func (s *Server) internal(w http.ResponseWriter, err error) {
    s.Logger.Error("internal error", "error", err.Error())
    writeError(w, http.StatusInternalServerError, "internal error")
}

func (s *Server) health(w http.ResponseWriter, r *http.Request) {
    writeJSON(w, http.StatusOK, map[string]string{"status": "ok"})
```

```
}

func (s *Server) head(w http.ResponseWriter, r *http.Request) {
    recs, err := s.Ledger.Snapshot()
    if err != nil {
        s.internal(w, err)
        return
    }
    h := map[string]any{"records": len(recs), "head": ""}
    if n := len(recs); n > 0 {
        h["head"] = recs[n-1].Digest
    }
    writeJSON(w, http.StatusOK, h)
}

func (s *Server) list(w http.ResponseWriter, r *http.Request) {
    after, err := intParam(r, "after", 0, 0, 1<<31-1)
    if err != nil {
        writeError(w, http.StatusBadRequest, err.Error())
        return
    }
    limit, err := intParam(r, "limit", 100, 1, 1000)
    if err != nil {
        writeError(w, http.StatusBadRequest, err.Error())
        return
    }
    recs, err := s.Ledger.Snapshot()
    if err != nil {
        s.internal(w, err)
        return
    }
    if after > len(recs) {
        after = len(recs)
    }
    recs = recs[after:]
    if len(recs) > limit {
        recs = recs[:limit]
    }
    writeJSON(w, http.StatusOK, map[string]any{"after": after, "
    records": recs})
}

func intParam(r *http.Request, name string, def, lo, hi int) (int,
    error) {
    v := r.URL.Query().Get(name)
    if v == "" {
        return def, nil
    }
    n, err := strconv.Atoi(v)
    if err != nil || n < lo || n > hi {
        return 0, fmt.Errorf("%s must be an integer from %d to %d",
```

```
    name, lo, hi)
    }
    return n, nil
}

func (s *Server) lookup(w http.ResponseWriter, r *http.Request) {
    path := r.URL.Query().Get("path")
    if path == "" {
        writeError(w, http.StatusBadRequest, "path is required")
        return
    }
    recs, err := s.Ledger.Snapshot()
    if err != nil {
        s.internal(w, err)
        return
    }
    x, err := index.Build(recs)
    if err != nil {
        s.internal(w, err)
        return
    }
    e, ok := x.Lookup(path)
    if !ok {
        writeError(w, http.StatusNotFound, "no ledger record mentions
    that path")
        return
    }
    writeJSON(w, http.StatusOK, e)
}

type judgment struct {
    ID          string `json:"id"`
    Disposition string `json:"disposition"`
    Receipt     string `json:"receipt"`
}

// admit runs a batch of records through the pipeline under the
   server's
// policy and writes one ledger record saying what happened and on
   whose
// request. A batch that fails validation is recorded too.
func (s *Server) admit(w http.ResponseWriter, r *http.Request) {
    p, _ := principalFrom(r.Context())
    dec := json.NewDecoder(http.MaxBytesReader(w, r.Body, MaxBody))
    dec.DisallowUnknownFields()
    var batch []pipeline.Record
    if err := dec.Decode(&batch); err != nil {
        var tooBig *http.MaxBytesError
        if errors.As(err, &tooBig) {
            writeError(w, http.StatusRequestEntityTooLarge, "body
   exceeds limit")
```

```
            return
        }
        writeError(w, http.StatusBadRequest, "invalid JSON: "+err.Error())
        return
    }

    inscribed := make([]pipeline.Inscribed, len(batch))
    for i, rec := range batch {
        inscribed[i] = pipeline.Inscribe(rec)
    }
    valid, err := pipeline.Validate(inscribed)
    if err != nil {
        detail := map[string]string{"stage": "validate", "error": err.Error(), "principal": p.Name}
        if _, lerr := s.Ledger.Append("admit-failed", "http", detail); lerr != nil {
            s.internal(w, lerr)
            return
        }
        writeError(w, http.StatusUnprocessableEntity, err.Error())
        return
    }

    admitted := pipeline.Admit(valid, s.Policy)
    var out []judgment
    for _, j := range admitted.Judgments() {
        out = append(out, judgment{j.Record.ID, string(j.Disposition), string(j.Receipt)})
    }
    counts := map[string]int{}
    detail := map[string]string{"records": strconv.Itoa(valid.Len()), "principal": p.Name}
    for d, n := range admitted.Counts() {
        counts[string(d)] = n
        detail[string(d)] = strconv.Itoa(n)
    }
    rec, err := s.Ledger.Append("admit", "http", detail)
    if err != nil {
        s.internal(w, err)
        return
    }
    writeJSON(w, http.StatusOK, map[string]any{"ledger_seq": rec.Seq, "judgments": out, "counts": counts})
}
```

Listing 16.1: The service: routing table, adjudication, and handlers.

16.2 Reading and Writing a Request

A handler receives a `ResponseWriter` and a pointer to a `Request`, and the rules for each are strict in ways that the types do not enforce. The status line and the headers must be set before the first byte of the body is written. Calling `WriteHeader` sets the status, and writing the body without having called it sends a status of 200, so a handler that discovers an error after it has begun to write cannot change the status any more. The helper `writeJSON` in the listing therefore sets the content type, sends the status, and only then encodes the body, and it ignores the error from the encoder because nothing useful can be done once the status is committed.

The body of a request is a stream and not a value, and it comes from a client that may be slow, hostile, or simply mistaken about how much it is sending. A handler that decodes the whole body into memory without a limit lets any client exhaust the memory of the process by sending a large one. The function `http.MaxBytesReader` wraps the body in a reader that fails once a limit is passed, and the admission handler sets the limit at one mebibyte. When the limit is passed the decoder's error is an `*http.MaxBytesError`, which the handler recognizes with `errors.As`, the technique of Chapter 4, and answers with status 413 instead of reporting a generic failure. The decoder is also told to refuse unknown fields, which is the strict choice weighed in the last exercise of Chapter 13. A client that misspells a field name is told so, and does not have its record accepted with the field silently dropped.

16.3 One Goroutine per Request

The server starts a goroutine for every connection and calls the handler in it. This is the model that makes Go servers simple to write, and it is the reason the concurrency of Part III is not optional here. Two requests can be inside `admit` at the same moment, and both will want to append to the ledger. The `Server` holds a `ledger.Writer`, so both appends go through the channel to the single goroutine that owns the file. The test `TestConcurrentClientsShareOneLedgerSafely` sends forty simultaneous writes while ten other clients read, and checks afterward that the ledger holds exactly forty records and that its chain verifies. It was run under the race detector.

The readers raise a point that Part III did not need. The handlers for the head of the ledger, for a listing of records, and for an index lookup all need the contents of the ledger. If a handler called `ledger.Load` directly, it would read the file while the owning goroutine might be in the middle of writing a line, and could see a partial record, which would fail to decode. The `Writer` therefore has a method, `Snapshot`, that sends a request of a different kind through the same channel. The owning goroutine handles it between appends, loads the file, and replies with the records. A reader sees the ledger as it stood between two appends, and never a half-written line. The test `TestSnapshotSeesCompletedAppendsAndNeverATornRecord` runs eight appenders and a repeated reader at once, verifies the chain of every snapshot, and checks that no snapshot is shorter than an earlier one. The cost is that every read loads the whole file and queues behind pending appends, which is acceptable for a teaching service and would be the first thing to change in a busy one. The protection extends only to readers inside this process. A command-line program that reads the file at the same moment has no

such guarantee, which is a case of the single-process limit that Chapter 12 stated and that Chapter 18 returns to.

16.4 Timeouts and Shutdown

A server that waits forever for a client is a server that a few slow clients can stop. The `http.Server` type has timeouts for reading the request headers, for reading the whole request, for writing the response, and for idle connections, and all of them default to no limit. The command sets each of them. Without them, a client that opens connections and sends nothing holds a goroutine and a socket per connection indefinitely, and the process eventually runs out of one or the other.

Stopping a service has the same structure as stopping the pool of Chapter 11, with requests in place of files. The command listens for the interrupt and terminate signals through `signal.NotifyContext`, which returns a context that is cancelled when one arrives. When the context ends, the command calls `Shutdown` on the server with a deadline. The method closes the listeners so that no new connections are accepted, waits for the requests in progress to finish, and returns once they have or the deadline passes. Only then does the command close the ledger writer, and the order is essential. A writer closed first would make every request still in flight fail with `ErrClosed` on its append, after the client had been told nothing, whereas a writer closed last has every accepted append written before the process exits. The transcript in Listing 17.3 shows the sequence at the end, with the messages `shutting down` and `stopped` and a clean exit status.

16.5 Testing Without a Network

The package `net/http/httptest` supplies what is needed to test a handler without a real network. `NewRecorder` gives a response writer that captures what the handler wrote, and `NewServer` starts a real server on a loopback port and returns its address. The tests of this chapter use the second form, because they want the whole path, including the router, the adjudication, and the concurrency of real connections. They are collected in the next chapter, which is concerned with what they establish.

Exercises

Exercise 16.1. Write a middleware function, one that takes an `http.Handler` and returns another, that records the status code and duration of every request, using a wrapper around the `ResponseWriter` that captures the status. Explain why the wrapper must forward `WriteHeader` and why a handler that never calls it needs a default.

Exercise 16.2. Remove the `ReadHeaderTimeout` from the server and write a program that opens many connections and sends only a few bytes on each. Report what happens to the number of goroutines in the server, and restore the timeout.

Exercise 16.3. Change the order of shutdown so that the writer is closed before the server. Start a client that posts continuously, stop the server with a signal, and report what the client observes.

Exercise 16.4. Replace the full-file read in `Snapshot` with a request that returns only the records after a given sequence number, and have the listing handler use it. State what the writer must now assume about the part of the file it does not read.

Chapter 17

Routing, Adjudication, and Admission

17.1 Three Questions Again

Chapter 8 separated three questions that a small program tends to answer in one place: whether a record is well formed, whether the actor is entitled to request the change, and whether the change is made durable. The service has a parallel set of three, and the temptation to merge them is just as strong, because a web framework makes it easy to write a single function that looks at the request, decides whether the caller may do what it asks, and does it. The service keeps them apart in its structure. *Routing* chooses which handler a request is for, from the method and the path. *Adjudication* decides whether the caller may use that handler. *Admission* is the pipeline of Part II, which decides what becomes of the content of a record, and it happens inside a handler, after the first two have been settled. The three have different inputs, different authorities, and different failure statuses, and the code is arranged so that each is a separate piece.

17.2 Routing Decides From the Request Line

```
package service

import (
    "net/http"
    "sort"
    "strings"
)

// Route is one entry of the routing table: a method and an exact path, the
// handler for them, and what the caller must be to use it.
type Route struct {
    Method  string
    Path    string
    Need    Need
    Handler http.Handler
```

```
}

// Router chooses a route from a method and a path. Match takes those two
// strings and nothing else, so it cannot consult headers, credentials, or the
// body, and routing cannot come to depend on who is asking.
type Router struct {
    routes []Route
}

// NewRouter builds a router over a table of routes.
func NewRouter(routes []Route) *Router { return &Router{routes: routes} }

// Match returns the route for the request line, or a status code explaining
// why there is none: 404 when no route has the path, 405 when some route has
// the path but none with the method, together with the allowed methods.
func (rt *Router) Match(method, path string) (route *Route, status int, allow []string) {
    for i := range rt.routes {
        r := &rt.routes[i]
        if r.Path != path {
            continue
        }
        if r.Method == method {
            return r, http.StatusOK, nil
        }
        allow = append(allow, r.Method)
    }
    if len(allow) > 0 {
        sort.Strings(allow)
        return nil, http.StatusMethodNotAllowed, allow
    }
    return nil, http.StatusNotFound, nil
}

func allowHeader(methods []string) string { return strings.Join(methods, ", ") }
```

Listing 17.1: A routing table that sees only the method and the path.

The router is a table of routes, each naming a method, an exact path, a handler, and the requirement a caller must meet. Its method `Match` takes two strings, the method and the path, and returns a route or a status code explaining the absence of one. The signature is the point. A function that is given only the method and the path cannot read a header, a credential, a query parameter, or the body, and so routing cannot come to depend on who is asking or what they sent. The compiler guarantees that much. That nothing else leaks

into the choice, for instance through a package variable, is a matter of reading the thirty lines of the function.

When no route matches, the router distinguishes two situations that clients need to tell apart. If no route has the path, the answer is 404. If some route has the path but none with the method, the answer is 405, accompanied by an `Allow` header that lists the methods the path does support, in sorted order so that the header is the same on every request. The test `TestRouterDecidesFromMethodAndPathOnly` checks a table of such cases, including a trailing slash, which is a different path and receives 404, since the table matches paths exactly.

17.3 Adjudication Is a Function of Two Values

```
package service

import (
    "crypto/subtle"
    "fmt"
    "net/http"
    "strings"
)

// Principal is who a request says it comes from, once the credential has
// been checked. It carries what the adjudicator needs and nothing else.
type Principal struct {
    Name     string
    CanWrite bool
}

// Tokens maps bearer tokens to principals.
type Tokens struct {
    entries []tokenEntry
}

type tokenEntry struct {
    token string
    p     Principal
}

// ParseTokens reads a comma-separated list of name=token:mode entries, where
// mode is "r" or "rw". Empty and duplicate tokens are rejected, because a
// token shared by two principals would make the credential ambiguous.
func ParseTokens(s string) (*Tokens, error) {
    t := &Tokens{}
```

```
    seen := map[string]bool{}
    for _, item := range strings.Split(s, ",") {
        item = strings.TrimSpace(item)
        if item == "" {
            continue
        }
        name, rest, ok := strings.Cut(item, "=")
        tok, mode, ok2 := strings.Cut(rest, ":")
        if !ok || !ok2 || name == "" || tok == "" || (mode != "r" && mode != "rw") {
            return nil, fmt.Errorf("token entry %q: want name=token:r or name=token:rw", name)
        }
        if seen[tok] {
            return nil, fmt.Errorf("token entry %q: token already assigned", name)
        }
        seen[tok] = true
        t.entries = append(t.entries, tokenEntry{tok, Principal{Name: name, CanWrite: mode == "rw"}})
    }
    if len(t.entries) == 0 {
        return nil, fmt.Errorf("no tokens configured")
    }
    return t, nil
}

// Authenticate extracts a bearer token from the request and looks it up.
// Every configured token is compared, in constant time, whether or not an
// earlier one matched, so that the time taken does not reveal how much of a
// guess was right.
func (t *Tokens) Authenticate(r *http.Request) (Principal, bool) {
    const prefix = "Bearer "
    h := r.Header.Get("Authorization")
    if !strings.HasPrefix(h, prefix) {
        return Principal{}, false
    }
    got := []byte(h[len(prefix):])
    var found Principal
    ok := false
    for _, e := range t.entries {
        if subtle.ConstantTimeCompare(got, []byte(e.token)) == 1 {
            found, ok = e.p, true
        }
    }
    return found, ok
}

```

```
// Need is what a route requires of the caller.
type Need int

const (
    NeedNothing Need = iota // anyone, with or without a credential
    NeedRead                // any authenticated principal
    NeedWrite               // a principal allowed to write
)

// Permit is the adjudication: whether this principal may do what the route
// requires. It is a pure function of its two arguments.
func Permit(p Principal, authenticated bool, n Need) (status int, ok bool) {
    switch n {
    case NeedNothing:
        return http.StatusOK, true
    case NeedRead:
        if !authenticated {
            return http.StatusUnauthorized, false
        }
        return http.StatusOK, true
    case NeedWrite:
        if !authenticated {
            return http.StatusUnauthorized, false
        }
        if !p.CanWrite {
            return http.StatusForbidden, false
        }
        return http.StatusOK, true
    }
    return http.StatusForbidden, false // an undeclared need is refused
}
```

Listing 17.2: Bearer tokens, principals, and the permission rule.

Adjudication has two steps, which the code keeps distinct. *Authentication* establishes who is calling. The service reads a bearer token from the `Authorization` header and looks it up in a list of configured tokens, each of which names a principal and says whether it may write. *Authorization*, the permission rule, asks whether that principal may do what the route requires. The function `Permit` takes the principal, a flag saying whether authentication succeeded, and the route's requirement, and returns a status and a verdict. It reads nothing else. A route's requirement is one of three values: nothing, which makes the route public, as the health check is; read, which needs any authenticated principal; and write, which needs one allowed to write. An undeclared requirement is refused, for the reason that the policy vocabulary of Chapter 7 fails closed on an undeclared verdict: a system that decides what is permitted should deny what it does not recognize.

Two details of authentication come from the standard library and are worth stating. The comparison of tokens uses `subtle.ConstantTimeCompare`, which takes the same time

whether the first byte or the last differs, so that a client cannot learn a token a byte at a time by measuring how long the server takes to refuse a guess. The loop also compares against every configured token and does not stop at the first match, so that the time does not reveal the position of the token in the list. The parsing of the configuration rejects an empty token and a token assigned to two principals, because a shared credential would make the identity of the caller ambiguous.

Proposition 17.1. *For every request that reaches a handler of the routing table,* `Permit` *returned success for the principal derived from the request and the requirement of the route that matched.*

Proof. Every request enters through `Server.ServeHTTP`. The only call that runs a route's handler is the last statement of that method, and it is reached only after `Match` has returned a route and `Permit` has returned success for that route's requirement. Any other path through the method returns after writing an error. □

The proof depends on facts about the code that the compiler does not check. The handlers are methods of `Server`, and another function in the package could call one directly and skip the check, and the value that `New` returns exposes no handler outside the package. The argument is therefore local to the package and holds as long as `ServeHTTP` remains the only way in. The tests supply evidence of the property at its most important point. A principal who may only read posts a valid batch and receives 403, an anonymous caller posts and receives 401, and in both cases the ledger afterward has no records, so the handler, which would have appended one, did not run.

17.4 Routing Comes First

The order of the steps in `ServeHTTP` is a decision with consequences. Because routing precedes adjudication, a request for a path that does not exist receives 404 whether or not it carries a valid credential, and a request with the wrong method receives 405 with an `Allow` header under every credential. The test `TestRoutingErrorsDoNotDependOnCredentials` checks both statements across four different credentials. The alternative, in which every request is authenticated before anything else, would answer unauthenticated requests with 401 even for paths that do not exist, which reveals nothing about the table. The service takes the first order because it makes the router's behaviour a function of the request line alone, and the cost is that an anonymous caller can learn which paths exist by observing which ones answer 401 or 403 and which answer 404. For a service whose routes are published this costs nothing, and for one whose routes are secret it would be a reason to reverse the order.

17.5 Admission Is a Different Judgment

```
$ golab serve -addr 127.0.0.1:18080 -deny bo &
listening on 127.0.0.1:18080
$ curl -s -w " [%{http_code}]\n" http://127.0.0.1:18080/healthz
{"status":"ok"}
```

```
 [200]
$ curl -s -w " [%{http_code}]\n" http://127.0.0.1:18080/v1/ledger/head
{"error":"Unauthorized"}
 [401]
$ curl -s -w " [%{http_code}]\n" -H 'Authorization: Bearer rsecret' http
    ://127.0.0.1:18080/v1/ledger/head
{"head":"","records":0}
 [200]
$ curl -s -w " [%{http_code}]\n" -H 'Authorization: Bearer rsecret' --
    data @batch.json http://127.0.0.1:18080/v1/admit
{"error":"Forbidden"}
 [403]
$ curl -s -w " [%{http_code}]\n" -H 'Authorization: Bearer wsecret' --
    data @batch.json http://127.0.0.1:18080/v1/admit
{"counts":{"admitted":1,"refused":1},"judgments":[{"id":"a","disposition
    ":"admitted","receipt":"1243
    f922ac053fd9630ac3226527913c9e84ec244418dfc47ec41f2882c972af"},{"id
    ":"b","disposition":"refused","receipt":"79
    d63a9261e40de63c9763a4d6d0acb0c3be2d5cd4dbe5485aa0d6bf3e7a8981"}],"
    ledger_seq":1}
 [200]
$ curl -s -w " [%{http_code}]\n" -X DELETE http://127.0.0.1:18080/v1/
    admit
{"error":"Method Not Allowed"}
 [405]
$ curl -s -w " [%{http_code}]\n" http://127.0.0.1:18080/v1/nothing
{"error":"Not Found"}
 [404]
$ curl -s -w " [%{http_code}]\n" -H 'Authorization: Bearer rsecret' http
    ://127.0.0.1:18080/v1/ledger/head
{"head":"3
    baa709af34022d6f369b65316d38bd5de028fc20deb7671b8940e6d51e7aef3","
    records":1}
 [200]
$ kill -TERM $PID
(server exit status 0)
listening on 127.0.0.1:18080
shutting down
stopped
$ golab audit
ledger holds: 1 records
$ cut -d, -f1,3,4,5 golab-ledger.jsonl
{"seq":1,"action":"admit","subject":"http","detail":{"admitted":"1"
```

Listing 17.3: A session with the service: refusal, admission, and shutdown.

The handler for `POST /v1/admit` runs only for a caller who may write, and what it does is the work of Part II. It decodes a batch of records, inscribes each, validates the batch as a whole, and admits the valid set under the policy the server was started with. The caller of the transcript uses the token of a principal who may write, and the server was started with the policy that denies the author `bo`. The response says that record `a` was

admitted and `b`, which was written by `bo`, was refused, and gives the receipt of each. The adjudication let the caller through, and the policy still refused one of the records, which shows the separation working: the caller is entitled to submit, and the content is judged on its own terms.

A batch that fails validation, for instance because a record refers to one that is not in the batch, is answered with status 422 and is recorded in the ledger as a failed admission, as the command-line version records its own failures in Chapter 8. The tests cover malformed JSON and unknown fields, which are answered with 400 and are not recorded, since no batch was ever formed, and an oversized body, which is answered with 413. The distinction between a request that never became a batch and a batch that failed validation is one a reader of the ledger can rely on.

The ledger record for an admission carries the name of the principal. The first limit stated in Chapter 6 was that the tool had no notion of identity, and here it acquires a limited one. What the ledger now records is the name attached to the token that made the request. It does not identify a person, since anyone holding the token is that principal, and the token travels as a plain header. The service as written speaks plain HTTP. A bearer token sent over a network that anyone can observe is observable, so the service belongs behind a terminator of transport encryption, or on a loopback interface, and the book does not claim more for it than that.

Refusals at the adjudication stage are not recorded. A request that receives 401 or 403 leaves no mark in the ledger. The omission is deliberate and has a cost. A ledger that recorded every refused request would let any anonymous client append to it at will, filling it with entries and making the cost of an attack the cost of the ledger's growth. The question of what the ledger should say about refusals, raised at the end of Chapter 6, therefore has two different answers according to who is refused. A refusal by policy, which concerns a record from an authenticated caller, is recorded as part of the admission. A refusal of the caller is not, and an operator who needs it would use the server's logs, which Part VI takes up.

Exercises

Exercise 17.1. Add a route for `DELETE` on the ledger and give it a requirement that no principal can meet. Show with a test that the handler is unreachable, and explain what the 403 reveals to a caller that a 404 would not.

Exercise 17.2. Reverse the order of routing and adjudication so that every request is authenticated first. Rewrite the two tests of this chapter that depend on the order, and state which property of the router is lost.

Exercise 17.3. Add a rate limit on failed authentication per remote address, recording the address and the count in memory. Discuss what the limit does to a legitimate client behind a shared address, and why the limit is state that a restart loses.

Exercise 17.4. Write a handler that records refused callers in the ledger but caps the recording at a fixed number per minute, writing a single summary record when the cap is exceeded. Decide what the summary should say and whether it can itself be used to harm the ledger.

Chapter 18

Containers and the Edge of the Process

18.1 What a Container Is

The service runs on the machine where it was started, with that machine's files, network, and users visible to it. A container changes what the process can see and use without changing the process itself. It is not a virtual machine, which runs its own kernel, but an ordinary process of the host that the kernel has been told to treat specially. Three features of the Linux kernel do most of the work, and the others, such as the image format and the tooling, are conveniences built on them.

The first is the *namespace*. A namespace gives a process its own view of one kind of global resource. A process in a new process-identifier namespace sees itself as process number 1 and cannot see the processes outside. A new host-name namespace lets it set a host name that the rest of the machine does not share. A new network namespace gives it an empty set of network interfaces, and a new mount namespace gives it a private table of mounted filesystems. The program `unshare` runs a command in new namespaces, and the transcript below was produced with it, on a host running a Linux kernel.

```
$ ps -o pid,comm | head -3
  PID COMMAND
    1 process_api
    2 kthreadd
$ unshare --pid --fork --mount-proc ps -o pid,comm
    PID COMMAND
      1 ps
$ readlink /proc/self/ns/pid; unshare --pid --fork readlink /proc/self/
   ns/pid
pid:[4026531836]
pid:[4026532264]
$ hostname; unshare --uts sh -c "hostname demo; hostname"; hostname
vm
demo
vm
$ unshare --net cat /proc/net/dev | tail -n +3 | cut -d: -f1
    lo
$ unshare --mount sh -c "mount -t tmpfs none /mnt && touch /mnt/inside
```

```
   && ls /mnt"; ls /mnt | grep -c inside || true
inside
0
$ sleep 30 & kill -TERM $!   # an ordinary process
terminated by SIGTERM
$ unshare --pid --fork sleep 30 &   # the same, as process 1 of a new
   PID namespace
still running after SIGTERM
terminated by SIGKILL
```

Listing 18.1: Namespaces demonstrated with `unshare`.

The transcript shows each kind in turn. In a new process namespace the listing of processes contains only the command itself, as number 1, and the namespace identifiers read from the process's own entry differ between the host and the child. In a new host-name namespace the name can be changed and the host's name is untouched. In a new network namespace the only interface is the loopback. In a new mount namespace a filesystem mounted inside is not visible outside, and the final count of matching entries on the host is zero. These are the isolations that container tools combine.

The last two entries of the transcript show a consequence that matters to the service. An ordinary process that receives the terminate signal and has not arranged to handle it is ended. The first process of a process namespace is treated differently by the kernel, which delivers to it only the signals for which it has installed a handler, and a signal that cannot be caught and the kill signal are the exceptions. The same command that was terminated outside the namespace was still running after the terminate signal inside it, and was ended only by the kill signal. A service that runs as the first process of a container and does not handle the terminate signal will ignore the request to stop that container tools send first, and will be killed after a grace period without having finished anything. The `serve` command installs a handler for the signal through `signal.NotifyContext`, as Chapter 16 described, and the behaviour shown here is the reason that handler is necessary and not merely tidy.

The second feature is the *control group*, which limits what a group of processes may consume. A control group can cap memory, processor time, and the number of processes, and the kernel enforces the cap by reclaiming memory, by throttling, or by refusing to create more processes. A namespace limits what a process can see, and a control group limits what it can use, and the two are independent. The third feature is the root filesystem. A container starts with a directory tree of its own, assembled from the layers of an image, as its view of the files. A container whose tree contains only one program and the files that program needs has nothing else to run, to read, or to be tricked into running.

18.2 A Static Binary

The last point suggests how little a Go service needs. A Go program compiled with the `CGO_ENABLED=0` setting contains its own runtime and its own implementation of the system calls it uses, and links to no shared library. The `file` command reports it as statically linked, and such a program runs in a directory tree that has no C library, no shell,

and no certificate store. The setting has a cost for programs that depend on the C library's behaviour, such as name resolution through the system's configured services, where the pure-Go implementation takes over, and for a service that does none of these things it costs nothing. The flags `-trimpath` and `-ldflags='-s -w'` remove the build directory's path and the symbol tables from the binary, which makes it smaller and its contents less dependent on where it was built.

```
# The binary is built outside and copied in, so the image contains
   nothing
# but the program. FROM scratch is an empty filesystem.
FROM scratch
COPY golab /golab
# An empty directory owned by that user; a volume mounted here starts
   out
# with the same ownership.
COPY --chown=10001:10001 data /data
# A numeric user, because the image has no /etc/passwd to look a name up
    in.
USER 10001:10001
EXPOSE 8080
# The ledger lives in a volume mounted at /data, not in the image.
WORKDIR /data
ENTRYPOINT ["/golab", "-ledger", "/data/golab-ledger.jsonl"]
CMD ["serve", "-addr", "0.0.0.0:8080"]
```

Listing 18.2: The image definition.

The definition in Listing 18.2 starts from `scratch`, which is an empty filesystem, copies in the binary, and adds an empty directory for data owned by a numeric user. The user is numeric because the image has no file of user names to look a name up in. The entry point runs the program with the ledger placed in the data directory, and the default command starts the service listening on all interfaces, since an address on loopback inside a container would be reachable only from inside it. The binary is built outside the image and copied in, by the script `deploy/build.sh`, which sets the build flags and then builds the image. A multi-stage definition that compiles inside a Go image is a common alternative and produces the same contents.

18.3 Running It

```
$ file build/golab | cut -d, -f1-4
build/golab: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
   statically linked
$ ls -l build/golab | cut -d" " -f5,9
6095032 build/golab
$ docker images golab:dev --format "{{.Repository}}:{{.Tag}} {{.Size}}"
golab:dev 8.72MB
```

```
$ docker run -d --name golab-svc --read-only --memory 64m --cpus 0.5 --
    cap-drop ALL --pids-limit 64 -v gdata:/data -e GOLAB_TOKENS=alice=
    wsecret:rw -p 18081:8080 golab:dev | cut -c1-12
4bb7c1848a06
$ docker logs golab-svc
listening on 0.0.0.0:8080
$ curl -s -w ' [%{http_code}]\n' -H 'Authorization: Bearer wsecret' --
    data '[{"id":"a","author":"ann","scope":"lab/x","body":"first"}]'
    http://127.0.0.1:18081/v1/admit | cut -c1-120
{"counts":{"admitted":1},"judgments":[{"id":"a","disposition":"admitted
    ","receipt":"1243f922ac053fd9630ac3226527913c9e84
 [200]
$ docker top golab-svc -o pid,user,comm
PID                 USER                COMMAND
8367                10001               golab
$ cat /sys/fs/cgroup/memory/docker/$(docker inspect -f "{{.Id}}" golab-
    svc)/memory.limit_in_bytes
67108864
$ docker exec golab-svc /golab audit
ledger holds: 1 records
$ time docker stop golab-svc
golab-svc

real    0m0.244s
user    0m0.013s
sys     0m0.038s
$ docker inspect -f "exit={{.State.ExitCode}}" golab-svc
exit=0
$ docker logs golab-svc
listening on 0.0.0.0:8080
shutting down
stopped
$ docker run --rm -v gdata:/data golab:dev audit
ledger holds: 1 records
```

Listing 18.3: Building and running the service in a container.

The transcript in Listing 18.3 follows the image from build to stop. The binary is a statically linked executable of about six megabytes, and the image built from it is under nine. The container is started with the root filesystem read-only, a memory limit of 64 megabytes, half a processor, all Linux capabilities dropped, a limit of 64 processes, and a named volume at the data directory. Each of those restricts the service further and none is needed for it to work. The service starts, logs the address it listens on, and answers an authenticated request to admit a record. The process listing from the host shows the program running as the numeric user, with a process number that belongs to the host's namespace and not the number 1 it has inside. Reading the control group's file for the memory limit on the host shows 67108864 bytes, the 64 megabytes that were asked for. That file's location is specific to the older version of control groups that this host runs, and a host on the newer version keeps the limit in a differently named file under a different path.

Stopping the container sends the terminate signal to the program, and the transcript

shows the result of the handler described in the previous chapter: the stop takes a quarter of a second, the exit code recorded by the container runtime is zero, and the log ends with the messages `shutting down` and `stopped`. Had the program ignored the signal, the stop would have waited out the runtime's grace period, ten seconds by default, and ended the process with the kill signal. The last command starts a second container from the same image against the same volume, runs the audit, and reads the same ledger with one record in it. The volume is what carries the data past the life of the container, and the container itself was already stopped when the audit ran.

18.4 What a Container Does Not Decide

A container packages a program and limits what it can see and consume. It does not decide what the program is permitted to do, and the separation matters more here than anywhere else in the book, because a container looks like a security decision and is only a deployment one. The token in the environment of the service in the transcript is what lets a caller write, and the container neither knows nor checks it. The limits on memory and processes bound the damage a defect or an abusive caller can do to the host, and they do not make a request legitimate. The restrictions on capabilities and on the root filesystem reduce what an attacker who compromises the program could do next, and they do not prevent the compromise. All containers on a host share its kernel, so a flaw in the kernel is a flaw in every container's isolation, which is the difference from a virtual machine that the first section began with.

The container also leaves the single-writer limit of Chapter 12 exactly where it was. The service is the sole owner of the ledger within its process, and it is the sole owner of the file only if nothing else appends to it. Two containers started from the image against the same volume would each have a writer on the same file, and they would fork the chain just as twenty command-line processes did. The audit container of the transcript only reads, and reads were the one thing a second process could do safely, up to the caveat in Chapter 16 that it might see a line mid-write. A deployment of this service is therefore a single instance, and scaling it by running more copies is not available without the file lock of the exercise in Chapter 12 or a different arrangement for the ledger altogether.

18.5 Where Part V Ends

The tool is now a service with a defined interface, a routing table that is a function of the request line, a permission rule that is a function of the caller and the route, an admission step that is the pipeline of Part II, a ledger owned by one goroutine, and a container that runs it with limits. Each layer has been given a statement of what it guarantees and what it does not, and Appendix C collects them. Part VI turns from building the tool to measuring it. It asks how fast the ledger and the pool are, what the service is doing while it runs, how the tool behaves when its disk or its network fails, and whether a compressed summary of a ledger can still support the decisions the full ledger could.

Exercises

Exercise 18.1. Run the service in a container that has no handler for the terminate signal, by removing the `signal.NotifyContext` call, and time the stop. Report the time and the exit code, and explain both in terms of the first process of a namespace.

Exercise 18.2. Start the container with the root filesystem writable and the data directory not mounted as a volume. Add a record, stop and remove the container, start another, and report what the ledger contains. State what this shows about where state belongs.

Exercise 18.3. Run two containers against the same volume, each posting to its own service, and report what the audit says afterward. Then propose an arrangement, short of a file lock, in which only one of them ever writes.

Exercise 18.4. Write a multi-stage image definition that compiles the program inside a Go build image and copies the result into the empty image. Compare the final image with the one built here, and say what the extra stage adds to the time of a build.

Part VI

Experimental Systems

Chapter 19

Measurement

19.1 Performance Claims Are Claims

Earlier chapters made statements about cost without measuring anything. Chapter 12 said that each append to the ledger reads and checks the whole file, so that the cost of n appends grows as n^2, and Chapter 14 said that an index saves work when it is extended more often than it is rebuilt. Both statements follow from reading the code, and both are the kind of statement that a measurement can confirm, refine, or contradict. Part VI treats the tool as an object of study, and the first instrument is the benchmark. A statement about speed deserves the same treatment as any other claim in the book: it should say what was measured, on what, with what result, and what the result does not show.

The `testing` package supports benchmarks as ordinary functions. A function whose name begins with `Benchmark` and which takes a `*testing.B` is run by `go test -bench`, and its body loops `b.N` times. The framework chooses `b.N`: it runs the function with a small value, measures, and raises the value until the total time is long enough to be reliable, then reports the time per iteration. Inside the function, `b.StopTimer` and `b.StartTimer` exclude setup work from the measurement, and `b.Run` creates sub-benchmarks, which is how one function measures a range of sizes. The flag `-benchmem` adds the number of bytes and allocations per iteration, and `b.SetBytes` lets the framework report throughput.

19.2 The Cost of an Append

```
package ledger

import (
    "encoding/json"
    "fmt"
    "os"
    "path/filepath"
    "testing"
    "time"
)
```

```
// buildLedger writes n records to a new file and returns its path.
func buildLedger(b *testing.B, n int) string {
    b.Helper()
    path := filepath.Join(b.TempDir(), "base.jsonl")
    f, err := os.Create(path)
    if err != nil {
        b.Fatal(err)
    }
    defer f.Close()
    var recs []Record
    now := time.Unix(1_700_000_000, 0)
    for i := 0; i < n; i++ {
        r, err := Extend(recs, "verify-file", fmt.Sprintf("papers/f%05d.txt", i%500), map[string]string{"sha256": "00"}, now)
        if err != nil {
            b.Fatal(err)
        }
        recs = append(recs, r)
        line, _ := json.Marshal(r)
        f.Write(append(line, '\n'))
    }
    return path
}

// BenchmarkAppendBySize measures one append to a ledger that already holds n
// records. The file is restored before each iteration, outside the timer, so
// the size stays at n.
func BenchmarkAppendBySize(b *testing.B) {
    for _, n := range []int{100, 1000, 10000} {
        b.Run(fmt.Sprintf("n=%d", n), func(b *testing.B) {
            base := buildLedger(b, n)
            data, _ := os.ReadFile(base)
            work := filepath.Join(b.TempDir(), "work.jsonl")
            now := time.Unix(1_700_000_000, 0)
            b.ResetTimer()
            for i := 0; i < b.N; i++ {
                b.StopTimer()
                os.WriteFile(work, data, 0o644)
                b.StartTimer()
                if _, err := Append(work, "run", "x", nil, now); err != nil {
                    b.Fatal(err)
                }
            }
        })
    }
}

```

```
type noSync struct{ *os.File }

func (noSync) Sync() error { return nil }

// BenchmarkSyncCost compares appending with and without the flush to stable
// storage, on a small ledger so that the fixed cost of the flush dominates.
func BenchmarkSyncCost(b *testing.B) {
    for _, mode := range []string{"sync", "nosync"} {
        b.Run(mode, func(b *testing.B) {
            old := openAppend
            defer func() { openAppend = old }()
            if mode == "nosync" {
                openAppend = func(p string) (appendFile, error) {
                    f, err := os.OpenFile(p, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0o644)
                    return noSync{f}, err
                }
            }
            base := buildLedger(b, 100)
            data, _ := os.ReadFile(base)
            work := filepath.Join(b.TempDir(), "work.jsonl")
            now := time.Unix(1_700_000_000, 0)
            b.ResetTimer()
            for i := 0; i < b.N; i++ {
                b.StopTimer()
                os.WriteFile(work, data, 0o644)
                b.StartTimer()
                if _, err := Append(work, "run", "x", nil, now); err != nil {
                    b.Fatal(err)
                }
            }
        })
    }
}
```

Listing 19.1: Benchmarks for appending to a ledger.

The benchmark of Listing 19.1 measures one append to a ledger that already holds n records, for n equal to a hundred, a thousand, and ten thousand. To keep n constant, it restores the file to its starting contents before every iteration, with the timer stopped, so that each measured append sees a ledger of exactly the stated size. The results come from one run on a machine with two processors.

```
$ go test -run xxx -bench . -benchmem ./internal/ledger ./internal/index
    ./internal/pool
goos: linux
goarch: amd64
pkg: golab/internal/ledger
```

```
cpu: Intel(R) Xeon(R) Processor @ 2.80GHz
BenchmarkAppendBySize/n=100-2                895           1348302 ns/op
             245344 B/op       2423 allocs/op
BenchmarkAppendBySize/n=1000-2               121          10585569 ns/op
            1768142 B/op      24029 allocs/op
BenchmarkAppendBySize/n=10000-2               12         101893199 ns/op
           19230172 B/op     240052 allocs/op
BenchmarkSyncCost/sync-2                     679           1740344 ns/op
             245331 B/op       2423 allocs/op
BenchmarkSyncCost/nosync-2                  1258            996877 ns/op
             245331 B/op       2423 allocs/op
goos: linux
goarch: amd64
pkg: golab/internal/index
cpu: Intel(R) Xeon(R) Processor @ 2.80GHz
BenchmarkBuild/n=1000-2                      332           3105787 ns/op
              662561 B/op       8519 allocs/op
BenchmarkBuild/n=10000-2                      33          34811688 ns/op
             5847500 B/op      80525 allocs/op
BenchmarkBuild/n=100000-2                      4         379772019 ns/op
            59127546 B/op     800528 allocs/op
BenchmarkApplyOne-2                     10335445               117.9 ns/
   op            10 B/op          0 allocs/op
goos: linux
goarch: amd64
pkg: golab/internal/pool
cpu: Intel(R) Xeon(R) Processor @ 2.80GHz
BenchmarkVerifyAll/workers=1-2                 5         254820501 ns/op
            263.36 MB/s     2136328 B/op        773 allocs/op
BenchmarkVerifyAll/workers=2-2                 8         144323960 ns/op
            464.99 MB/s     2137198 B/op        775 allocs/op
BenchmarkVerifyAll/workers=4-2                 8         132528408 ns/op
            506.37 MB/s     2136908 B/op        777 allocs/op
BenchmarkVerifyAll/workers=8-2                10         116525809 ns/op
            575.91 MB/s     2138318 B/op        782 allocs/op
```

Listing 19.2: Benchmarks of the ledger, the index, and the pool.

The three results for appending are 1.35 milliseconds, 10.6 milliseconds, and 102 milliseconds. Each tenfold increase in the size of the ledger multiplies the time by roughly ten, and the number of allocations grows in the same proportion, from about two thousand to about two hundred and forty thousand. The cost of one append is proportional to the number of records already present, which is the behaviour Chapter 12 predicted. The cost per record is about ten microseconds on this machine. If the trend continued, a ledger of a million records would take about ten seconds to append to once, and a ledger of a million records built by a million appends would have cost the sum, which is quadratic. The extrapolation is a statement about a straight line drawn through three points. It is not a measurement of a ledger of a million records, and the measurements here establish only that the growth is linear over the range from a hundred to ten thousand.

A benchmark says how much and not why. The profiler says where.

19.3 Where the Time Goes

The standard toolchain can sample the running program and report which functions the samples fall in. The flag `-cpuprofile` on `go test` writes a profile, and `go tool pprof` reads it. The report distinguishes *flat* time, spent in a function's own instructions, from *cumulative* time, spent in the function and everything it calls, and for finding the structure of a cost the cumulative column is the useful one.

```
$ go test -run xxx -bench AppendBySize/n=10000 -benchtime 40x -
   cpuprofile cpu.out -o ledger.test ./internal/ledger
$ go tool pprof -top -cum ledger.test cpu.out    # selected rows
Duration: 4.24s, Total samples = 4.21s (99.38%)
      flat  flat%   sum%        cum   cum%
         0     0%     0%      3.44s 81.71%  golab/internal/ledger.Append
         0     0%     0%      2.32s 55.11%  golab/internal/ledger.Load
     0.01s  0.24%  0.24%      2.03s 48.22%  encoding/json.Unmarshal
         0     0%  6.41%      1.16s 27.55%  golab/internal/ledger.digest
         0     0%  6.41%      1.12s 26.60%  golab/internal/ledger.Check
     0.08s  1.90%  8.31%      0.73s 17.34%  runtime.mallocgc
     0.02s  0.48%  8.79%      0.67s 15.91%  encoding/json.Marshal
     0.02s  0.48% 25.18%      0.31s  7.36%  runtime.
   mallocgcSmallScanNoHeader
     0.02s  0.48% 38.00%      0.17s  4.04%  runtime.mallocgcSmallNoscan
```

Listing 19.3: Selected rows of a CPU profile of appending to a ledger of ten thousand records.

The profile in Listing 19.3 says that `Append` accounts for about four fifths of all the samples, the remainder being the restoration of the file between iterations. Within `Append`, loading the ledger takes the larger share, 55 percent of all samples, and nearly all of it is spent decoding JSON. Checking the chain takes 27 percent, and almost all of that is recomputing digests. The write of the new record does not appear among the visible costs. The time is going into reading and verifying the ledger that is already there, which is the work that the single-writer design of Chapter 12 was said to repeat. A writer that kept the head digest and the next sequence number in memory, and checked the file only when it opened it, would remove both of the large items from the per-append cost. That change is the third exercise of Chapter 12, and it has a price, since the writer then trusts its own memory between appends. The measurements identify what could be saved and say nothing yet about whether the saving is worth the price.

19.4 The Flush to Stable Storage

Part VI adds one thing to `Append`, a call to `Sync` on the file before it is closed, so that the record is on stable storage when the function returns and not merely in the operating system's cache. The second benchmark compares an append with and without the call, on a ledger of a hundred records, where the fixed costs dominate. The result is 1.74 milliseconds with the flush and 1.00 millisecond without, so the flush adds about 0.74 milliseconds to

each append on this machine. That number belongs to this machine's storage and not to the program. A different device would give a different figure, and a filesystem in memory would give almost none. The decision to flush is a decision about durability: without it, a crash can lose records that were reported as written. For a ledger whose purpose is to be relied on after the fact, the book takes the cost, and the benchmark makes the size of that cost visible, which a decision of this kind should have.

19.5 The Index and the Pool

The index benchmarks measure two things. Building an index from a whole ledger in memory takes 3.1 milliseconds for a thousand records, 34.8 for ten thousand, and 380 for a hundred thousand, which is between three and four microseconds per record and again linear. Applying a single record to an existing index takes about 118 nanoseconds and allocates nothing. The ratio between the two figures is large, and it needs care. The cost of 118 nanoseconds is the cost of `Apply` alone. The method `CatchUp`, which is what `Open` calls, checks the whole chain before applying new records, as Chapter 14 noted, and a check of the chain is most of what a build costs. In the present code, loading and extending an index is therefore not as cheap as the figure for `Apply` suggests, and the fourth exercise of Chapter 14, which proposes checking only the new records, is where that saving would come from.

The pool benchmark hashes sixty-four files of one mebibyte with one, two, four, and eight workers. The times fall from 255 milliseconds to 144, 133, and 117, and the throughput rises from 263 to 465, 506, and 576 megabytes per second. The machine has two processors, and the step from one worker to two gives a speedup of 1.8, which is what a CPU-bound task on two processors should give. The further gains from four and eight workers are smaller, and the figure for eight is above what two processors could deliver for pure computation. The measurement does not say why. It may reflect time the workers spend waiting on the operating system, which other workers can overlap, or noise in a single run on a shared machine. A claim about the cause would need further experiments, and the measurement supports only the statement that more workers than processors still helped a little in this run.

19.6 What a Benchmark Cannot Say

Every number in this chapter comes from one run on one virtual machine whose other tenants are unknown. The benchmark framework reports a mean and no spread. A responsible comparison repeats each benchmark several times with the `-count` flag and compares the distributions, with a tool such as `benchstat` from the Go project, and a single figure cannot show whether a difference between two figures is larger than the run-to-run variation. The files in the pool benchmark fit in the operating system's cache after the first pass, so the figures describe hashing and scheduling and say nothing about a cold disk. The environment that produced the numbers is printed at the head of the benchmark output, with the operating system, the architecture, and the processor, and it belongs next to any

figure that is quoted. A number without its environment is not a measurement of anything in particular.

Exercises

Exercise 19.1. Implement the writer that holds the head digest and the next sequence number in memory, and repeat the benchmark of this chapter for ledgers of a hundred, a thousand, and ten thousand records. Report the new figures, the new profile, and the new risks that the cache introduces.

Exercise 19.2. Run the benchmark of the pool ten times with the `-count` flag and report the range of each figure. State which of the conclusions of this chapter survive the variation and which do not.

Exercise 19.3. Write a benchmark of `ledger.Load` alone for the three sizes, and compare its time with the share the profile assigned to loading. Explain any discrepancy.

Exercise 19.4. Replace the call to `Sync` with a policy that flushes once every hundred appends or every second, whichever comes first. Describe exactly what a crash can now lose, and decide whether a ledger with such a policy can still make the statement that a record reported as written is durable.

Chapter 20

Logs and Faults

20.1 Two Records With Different Jobs

The service now keeps two kinds of record, and they should not be confused. The ledger is the durable account of what the tool decided and did. It is chained, it is checked, and it refuses to grow when damaged. A log is an operational stream, written for the person who runs the service so that they can see what is happening and diagnose what went wrong. Nothing in the log is chained, lines can be lost when a process dies, and the log is not evidence of anything in the sense the ledger tries to be. The distinction decides what goes where. A refusal of an anonymous caller does not belong in the ledger, for the reason given in Chapter 17, that anyone could fill the ledger by being refused, and it does belong in the log, which can be rotated and discarded.

The standard library's `log/slog` package writes structured entries, each with a level, a message, and a list of named values, and its handlers render them as JSON lines or as readable text. The service holds a `*slog.Logger`, and `ServeHTTP` writes one entry per request after the handler has finished, whatever the outcome, with the method, the path, the status, the number of bytes sent, the duration in milliseconds, and the name of the principal. To know the status after the fact, the method wraps the response writer in a small recorder that remembers the status that was sent, and the recorder implements the `Unwrap` method so that code reaching for optional features of the underlying writer, such as flushing, can still find it. The level follows the status: server errors are logged as errors, client errors as warnings, and everything else as information.

```
$ golab serve -addr 127.0.0.1:18082 2> requests.log &
$ curl -s -o /dev/null -H 'Authorization: Bearer rsecret' http
   ://127.0.0.1:18082/v1/ledger/head
$ curl -s -o /dev/null -H 'Authorization: Bearer rsecret' --data '[]'
   http://127.0.0.1:18082/v1/admit
$ curl -s -o /dev/null -H 'Authorization: Bearer wrong' http
   ://127.0.0.1:18082/v1/ledger/head
$ cat requests.log
{"time":"2026-10-25T15:45:11.196759277-03:00","level":"INFO","msg":"
   request","method":"GET","path":"/v1/ledger/head","status":200,"bytes
   ":24,"duration_ms":0.124,"principal":"bob"}
```

```
{"time":"2026-10-25T15:45:11.210047644-03:00","level":"WARN","msg":"
   request","method":"POST","path":"/v1/admit","status":403,"bytes
   ":22,"duration_ms":0.03,"principal":"bob"}
{"time":"2026-10-25T15:45:11.222823769-03:00","level":"WARN","msg":"
   request","method":"GET","path":"/v1/ledger/head","status":401,"bytes
   ":25,"duration_ms":0.016,"principal":""}
```

Listing 20.1: Three requests and the entries they produced.

The entries of Listing 20.1 show the three cases. A reader may fetch the head of the ledger, a reader who tries to write is refused with status 403, and a caller with an unknown token is refused with 401, with an empty principal because authentication failed. The refusals appear at the warning level, which is where an operator looks for them, and nothing about them reaches the ledger. The entries contain no credential. That is a property worth testing and not merely intending, because a log is the place where secrets leak: the test sends requests with a valid token and an invalid one and checks that neither string appears in the output, nor the word naming the authentication scheme. The query string is also left out of the entry, since a caller might put anything in it. What is logged is chosen, field by field, and the default is to leave a value out.

The same discipline applies to what the service tells the client. The handlers that fail because of the server's own machinery, such as a ledger that cannot be written, now answer with the status 500 and the words `internal error`, and send the cause to the log. A message from the operating system names file paths, and a client has no use for the layout of the server's disk. The test of this behaviour points the writer at a directory that does not exist, posts a batch, and checks that the response contains no part of the path while the log entry does.

20.2 Faults as Inputs

A program's behaviour on success is the part its author thinks about. Its behaviour when the machine fails to cooperate is mostly unexamined until it happens, and the failures that matter most to a ledger, a full disk, a write that does not complete, a flush that fails, are the ones that tests do not produce on their own. Fault injection is the practice of producing them on purpose. There are two ways to do it, and the chapter uses both. A simulated fault replaces part of the environment with something that fails in a chosen way, and it is repeatable and fast. A real fault arranges for the environment itself to fail, and it is more convincing and more awkward.

The real fault used here is a full filesystem. In a private mount namespace, as in Chapter 18, the experiment mounts a small memory-backed filesystem of eight kibibytes and appends records to a ledger on it until an append fails. The ledger as it stood at the end of Part V behaved as follows.

```
$ mount -t tmpfs -o size=8k none /mnt/small   # a filesystem that holds
   8192 bytes
$ # append until the filesystem is full, with the ledger as it was
   before the repair
```

```
appends that succeeded: 29
error from the failing append:
golab: write ledger: write /mnt/small/l.jsonl: no space left on device
ledger size in bytes: 8192
$ golab audit
golab: ledger line 30: unexpected end of JSON input
$ # the same, with the repaired Append
appends that succeeded: 29
error from the failing append:
golab: write ledger: write /mnt/small/l.jsonl: no space left on device
ledger size in bytes: 8134
$ golab audit
ledger holds: 29 records
```

Listing 20.2: Appending until the filesystem is full, before and after the repair.

In the first half of Listing 20.2 the thirtieth append failed with the error that the device had no space, which is the correct report. But the file was found to be exactly 8192 bytes, the whole of the filesystem, and the audit failed with an error about line 30, ending unexpectedly. The write of the thirtieth record had succeeded in part. The first bytes of the line fitted in the last block of the file, the rest did not, and the operating system returned an error after storing the fragment. The ledger now ended in half a record. Every later command that loaded the ledger failed, so the failure of one append had destroyed the usability of the whole ledger. The protection built into `Append`, which refuses to extend a broken chain, did what it was designed to do, and what it protected was a ledger that the failed append had itself broken.

The repair, in the second half of the listing, is to remember the length of the file before writing and to cut the file back to that length if the write does not complete. The same experiment now ends with the same twenty-nine successful appends and the same error on the thirtieth, a file of 8134 bytes, and an audit that finds twenty-nine records and a valid chain. The failed append left no trace in the file. The fix is a few lines in the function that writes the line.

20.3 Seams and Simulated Faults

The real fault cannot be part of the ordinary test suite, since it needs privileges and a mount, so the repair has tests of the simulated kind. The ledger package declares a small interface, `appendFile`, naming the five methods of an open file that the writing of a record uses, and a package-level variable, `openAppend`, that returns a value of that type. In normal use the variable opens a real file. A test replaces it with a function that returns a wrapper around a real file, which can be told to write only part of a line and then fail, to refuse to truncate, or to fail when synchronizing.

```
1 package ledger
2
3 import (
4     "errors"
```

```
    "os"
    "path/filepath"
    "strings"
    "testing"
    "time"
)

// faultyFile wraps a real file and fails in chosen ways.
type faultyFile struct {
    *os.File
    writeOnly   int // write this many bytes of the line, then fail; -1 for no fault
    failTrunc   bool
    failSync    bool
    truncations int
}

var errDisk = errors.New("no space left on device")

func (f *faultyFile) Write(b []byte) (int, error) {
    if f.writeOnly < 0 {
        return f.File.Write(b)
    }
    n := f.writeOnly
    if n > len(b) {
        n = len(b)
    }
    f.File.Write(b[:n])
    return n, errDisk
}

func (f *faultyFile) Truncate(n int64) error {
    f.truncations++
    if f.failTrunc {
        return errors.New("truncate refused")
    }
    return f.File.Truncate(n)
}

func (f *faultyFile) Sync() error {
    if f.failSync {
        return errors.New("sync refused")
    }
    return f.File.Sync()
}

func TestPartialWriteIsCutBackAndLedgerStaysUsable(t *testing.T) {
    path := filepath.Join(t.TempDir(), "l.jsonl")
    now := time.Unix(1_700_000_000, 0)
    if _, err := Append(path, "ok", "first", nil, now); err != nil {
        t.Fatal(err)
```

```
    }
    before, _ := os.ReadFile(path)

    var ff *faultyFile
    old := openAppend
    openAppend = func(p string) (appendFile, error) {
        f, err := os.OpenFile(p, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0o644)
        ff = &faultyFile{File: f, writeOnly: 40}
        return ff, err
    }
    _, err := Append(path, "ok", "second", nil, now)
    openAppend = old
    if !errors.Is(err, errDisk) {
        t.Fatalf("err = %v, want the disk error", err)
    }
    if ff.truncations != 1 {
        t.Fatalf("truncations = %d", ff.truncations)
    }
    after, _ := os.ReadFile(path)
    if string(after) != string(before) {
        t.Fatalf("file changed by a failed append:\n%q\n%q", before, after)
    }
    if _, err := Append(path, "ok", "third", nil, now); err != nil {
        t.Fatalf("ledger unusable after a failed append: %v", err)
    }
    recs, err := Load(path)
    if err != nil || len(recs) != 2 || Check(recs) != nil {
        t.Fatalf("len=%d err=%v", len(recs), err)
    }
}

func TestFailureToRemoveTheFragmentIsReported(t *testing.T) {
    path := filepath.Join(t.TempDir(), "l.jsonl")
    old := openAppend
    defer func() { openAppend = old }()
    openAppend = func(p string) (appendFile, error) {
        f, err := os.OpenFile(p, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0o644)
        return &faultyFile{File: f, writeOnly: 10, failTrunc: true}, err
    }
    _, err := Append(path, "ok", "x", nil, time.Unix(1, 0))
    if err == nil || !errors.Is(err, errDisk) {
        t.Fatalf("err = %v", err)
    }
    if got := err.Error(); !strings.Contains(got, "could not be removed") {
        t.Fatalf("message hides the damage: %q", got)
    }
```

```
    if _, lerr := Load(path); lerr == nil {
        t.Fatal("expected the fragment to be visible to Load")
    }
}

// An error from Append does not mean the record is absent. After a failed
// sync the record is in the file, and the caller has been told of a failure.
func TestSyncFailureReportsAnErrorButTheRecordIsPresent(t *testing.T) {
    path := filepath.Join(t.TempDir(), "l.jsonl")
    old := openAppend
    defer func() { openAppend = old }()
    openAppend = func(p string) (appendFile, error) {
        f, err := os.OpenFile(p, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0o644)
        return &faultyFile{File: f, writeOnly: -1, failSync: true}, err
    }
    if _, err := Append(path, "ok", "x", nil, time.Unix(1, 0)); err == nil {
        t.Fatal("expected an error")
    }
    recs, err := Load(path)
    if err != nil || len(recs) != 1 {
        t.Fatalf("len=%d err=%v", len(recs), err)
    }
}

func TestExtendMatchesAppend(t *testing.T) {
    path := filepath.Join(t.TempDir(), "l.jsonl")
    now := time.Unix(1_700_000_000, 0)
    var mem []Record
    for _, a := range []string{"a", "b", "c"} {
        disk, err := Append(path, a, "s", map[string]string{"k": a}, now)
        if err != nil {
            t.Fatal(err)
        }
        r, err := Extend(mem, a, "s", map[string]string{"k": a}, now)
        if err != nil {
            t.Fatal(err)
        }
        if r.Digest != disk.Digest || r.Seq != disk.Seq || r.Prev != disk.Prev {
            t.Fatalf("Extend and Append disagree: %+v %+v", r, disk)
        }
        mem = append(mem, r)
    }
    if err := Check(mem); err != nil {
```

```
            t.Fatal(err)
        }
}
```

Listing 20.3: Simulated faults for the ledger.

A seam of this kind costs something, and the cost should be stated. A package-level variable that tests replace is shared state, so two tests that replace it cannot run in parallel, and a test that forgets to restore it affects the tests that follow. The interface is wider than the code strictly needs, and it exists only for the tests. The alternative is to pass the opener as a parameter to every function, which changes every caller for the sake of a test. Each choice trades something, and the book takes the variable because the ledger has one write path and the number of tests that touch it is small. A larger system would likely choose differently.

The first test makes the wrapper write forty bytes of a record and fail, then checks three things: that `Append` returned the disk error, that the file's bytes are identical to what they were before the call, and that a later append succeeds and the ledger loads with a valid chain. The second makes the truncation fail as well, and checks that the error reports both failures, saying that the partial record could not be removed, and that the damaged ledger is visible to `Load`, which is the honest outcome when the damage cannot be undone. The third test is the most important for the contract of the function.

20.4 An Error Is Not a Statement That Nothing Happened

The third test makes the synchronization fail after a complete write. `Append` returns an error, since it cannot promise that the record is on stable storage, and the record is in the file. The test checks that the ledger contains it. A function that returns an error has told its caller that something went wrong, and the usual reading is that the operation did not take effect. That reading is false here, and it is false for a large class of operations that end with a step that can fail after the effect has occurred. A caller that retries after such an error will write the record twice.

The documentation of `Append` says so in a sentence: an error means that the record is not guaranteed to be present, and not that it is absent. The service passes the consequence on to its clients in the only way it can, by answering a failed append with a failure status and recording the cause in the log. A client that receives that status does not know whether its batch was recorded. The design that would let a client know is one in which the client supplies an identifier for the request and the server remembers which identifiers it has completed, so that a retry can be recognized, and the fourth exercise asks for it.

The repair also has a limit. If the process or the machine stops between the partial write and the truncation, the fragment remains, and the next command to load the ledger will report the broken line and stop. This is the loud failure and not the silent one, since the ledger refuses to proceed and does not pretend. Repairing such a ledger by cutting the torn final line is a decision about trust, because the same cut could hide the deletion of a real record, and the third exercise asks for a command that makes the cut only when everything before it verifies.

Exercises

Exercise 20.1. Write a test that stops a goroutine in the middle of `writeLine`, between the write and the truncation, by using the wrapper to block, and then loads the ledger from another goroutine. Report what the loader sees and explain why the single-writer design of Chapter 12 does not prevent it.

Exercise 20.2. Add a fault to the wrapper that makes `Close` fail after a successful write and flush. Decide what `Append` should report, and whether the record can be said to be absent, present, or unknown.

Exercise 20.3. Write a command `repair` that removes a final line of the ledger that does not decode, only if the records before it form a valid chain, and records the repair in the ledger. Discuss what a repair that hides the removal of a valid record would look like, and what the command must report to avoid it.

Exercise 20.4. Add an optional request identifier to the admission endpoint. Make the server remember, for a bounded time, the identifiers it has completed and the response it gave, and answer a repeated identifier with the stored response. State what the server must do about the identifier of a request whose append returned an error.

Chapter 21

Manifests and Reproducibility

21.1 What a Result Depends On

A figure from an experiment is a statement about a particular run, and a reader who wants to rely on it needs to know what the run consisted of. At least three things determine a result: the inputs it was computed from, the settings that were chosen, and the program that did the computing. A fourth, the machine and its state, matters for some results and not for others. The benchmarks of the previous chapter depend on all four, which is why their output opens with a description of the machine. An experiment whose result does not depend on timing, such as the one of the next chapter, depends only on the first three, and the first three can be written down.

A *manifest* is that written record. The manifest of `golab` lists the experiment's name and its parameters, the version of the Go toolchain that built the program, the module path, the build settings that the toolchain embedded in the binary, and for each input file its path, its size, and its SHA-256 digest. The package that builds it follows two rules that make it useful as evidence. It contains no clock reading and no machine name, so that two runs under the same conditions produce byte-identical manifests, and the paths of inputs are recorded relative to the input directory, so that moving the files does not change the manifest. The time at which a manifest was made is recorded by the ledger entry that refers to it, which carries the manifest's digest. The manifest is then a function of what it describes, and the ledger supplies the fact that the description was made at a certain point in the sequence.

```
1 // Package manifest records what an experiment ran on: the digests of
      its
2 // input files, its parameters, and the build of the program that ran
      it. Two
3 // runs with equal manifests used the same inputs and the same build,
      which is
4 // a precondition for comparing their results and does not by itself
     make the
5 // results equal.
6 //
7 // A manifest contains no clock reading and no host name. The time of
      a run is
```

```
// recorded by the ledger entry that refers to the manifest, so the manifest
// is a function of its inputs and the same inputs always give the same bytes.
package manifest

import (
    "crypto/sha256"
    "encoding/hex"
    "encoding/json"
    "fmt"
    "path/filepath"
    "runtime/debug"
    "sort"

    "golab/internal/artifact"
)

// Schema is the version of the manifest layout.
const Schema = 1

// Input is one input file, named relative to the input directory.
type Input struct {
    Path   string `json:"path"`
    SHA256 string `json:"sha256"`
    Size   int64  `json:"size"`
}

// Manifest describes one experiment run.
type Manifest struct {
    Schema   int               `json:"schema"`
    Name     string            `json:"name"`
    Params   map[string]string `json:"params"`
    Go       string            `json:"go"`
    Module   string            `json:"module"`
    Settings map[string]string `json:"settings"`
    Inputs   []Input           `json:"inputs"`
}

// buildInfo reads the facts the Go toolchain embedded in the running binary.
// It reports unknown values as empty strings when the binary carries none.
func buildInfo() (goVersion, module string, settings map[string]string) {
    settings = map[string]string{}
    info, ok := debug.ReadBuildInfo()
    if !ok {
        return "", "", settings
    }
    for _, s := range info.Settings {
```

```
        settings[s.Key] = s.Value
    }
    return info.GoVersion, info.Main.Path, settings
}

// Collect builds the manifest of an experiment whose inputs are the files
// under dir. An empty dir means the experiment has no input files.
func Collect(name string, params map[string]string, dir string) (Manifest, error) {
    goVersion, module, settings := buildInfo()
    m := Manifest{
        Schema:   Schema,
        Name:     name,
        Params:   copyMap(params),
        Go:       goVersion,
        Module:   module,
        Settings: settings,
        Inputs:   []Input{},
    }
    if dir == "" {
        return m, nil
    }
    found, err := artifact.Discover(dir)
    if err != nil {
        return Manifest{}, err
    }
    for _, d := range found {
        v, err := artifact.Verify(d)
        if err != nil {
            return Manifest{}, fmt.Errorf("input %s: %w", d.Path, err)
        }
        rel, err := filepath.Rel(dir, d.Path)
        if err != nil {
            return Manifest{}, err
        }
        m.Inputs = append(m.Inputs, Input{Path: filepath.ToSlash(rel), SHA256: v.SHA256(), Size: v.Size()})
    }
    sort.Slice(m.Inputs, func(i, j int) bool { return m.Inputs[i].Path < m.Inputs[j].Path })
    return m, nil
}

func copyMap(in map[string]string) map[string]string {
    out := make(map[string]string, len(in))
    for k, v := range in {
        out[k] = v
    }
    return out
```

```
}

// Encode returns the canonical bytes of the manifest. Maps are written with
// sorted keys and the inputs are already sorted, so equal manifests have
// equal encodings.
func (m Manifest) Encode() ([]byte, error) { return json.MarshalIndent(m, "", " ") }

// Digest is the SHA-256 of the canonical encoding.
func (m Manifest) Digest() (string, error) {
    b, err := json.Marshal(m)
    if err != nil {
        return "", err
    }
    sum := sha256.Sum256(b)
    return hex.EncodeToString(sum[:]), nil
}

// Decode reads a manifest and rejects other schema versions.
func Decode(b []byte) (Manifest, error) {
    var m Manifest
    if err := json.Unmarshal(b, &m); err != nil {
        return Manifest{}, err
    }
    if m.Schema != Schema {
        return Manifest{}, fmt.Errorf("manifest schema %d, want %d", m.Schema, Schema)
    }
    return m, nil
}

// Differences lists, in a fixed order, how b departs from a. An empty result
// means the manifests are equal.
func Differences(a, b Manifest) []string {
    var out []string
    if a.Name != b.Name {
        out = append(out, fmt.Sprintf("name: %q -> %q", a.Name, b.Name))
    }
    if a.Go != b.Go {
        out = append(out, fmt.Sprintf("go: %s -> %s", a.Go, b.Go))
    }
    if a.Module != b.Module {
        out = append(out, fmt.Sprintf("module: %s -> %s", a.Module, b.Module))
    }
    out = append(out, diffMaps("param", a.Params, b.Params)...)
    out = append(out, diffMaps("setting", a.Settings, b.Settings)...)
```

```
144
145     ai, bi := map[string]Input{}, map[string]Input{}
146     paths := map[string]bool{}
147     for _, x := range a.Inputs {
148         ai[x.Path], paths[x.Path] = x, true
149     }
150     for _, x := range b.Inputs {
151         bi[x.Path], paths[x.Path] = x, true
152     }
153     var names []string
154     for p := range paths {
155         names = append(names, p)
156     }
157     sort.Strings(names)
158     for _, p := range names {
159         x, inA := ai[p]
160         y, inB := bi[p]
161         switch {
162         case inA && !inB:
163             out = append(out, "input removed: "+p)
164         case !inA && inB:
165             out = append(out, "input added: "+p)
166         case x != y:
167             out = append(out, "input changed: "+p)
168         }
169     }
170     return out
171 }
172
173 func diffMaps(kind string, a, b map[string]string) []string {
174     keys := map[string]bool{}
175     for k := range a {
176         keys[k] = true
177     }
178     for k := range b {
179         keys[k] = true
180     }
181     var names []string
182     for k := range keys {
183         names = append(names, k)
184     }
185     sort.Strings(names)
186     var out []string
187     for _, k := range names {
188         av, inA := a[k]
189         bv, inB := b[k]
190         switch {
191         case inA && !inB:
192             out = append(out, fmt.Sprintf("%s removed: %s", kind, k))
193         case !inA && inB:
194             out = append(out, fmt.Sprintf("%s added: %s", kind, k))
```

```
            case av != bv:
                out = append(out, fmt.Sprintf("%s %s: %q -> %q", kind, k,
        av, bv))
            }
        }
        return out
}
```

Listing 21.1: The manifest of an experiment.

21.2 Build Information

The toolchain embeds facts about a build in the binary, and the function `debug.ReadBuildInfo` reads them back. The record includes the version of the toolchain, the path of the main module, and a list of key and value pairs for the settings of the build: the compiler, the target operating system and architecture, whether the C compiler was enabled, the level of the target instruction set, and, when the source is in a version-control repository, the revision and whether the working tree had uncommitted changes. The manifest copies these into its own fields.

```
$ golab manifest -name checks -params n=2,mode=full -out checks.manifest
    .json papers
manifest checks: 2 inputs, digest 3
    c7fff29c42ce866b73a782ee03b8f41692bcf292d944c2ba8efde3765cac09e
$ cat checks.manifest.json
{
 "schema": 1,
 "name": "checks",
 "params": {
  "mode": "full",
  "n": "2"
 },
 "go": "go1.24.7",
 "module": "golab",
 "settings": {
  "-buildmode": "exe",
  "-compiler": "gc",
  "CGO_CFLAGS": "",
  "CGO_CPPFLAGS": "",
  "CGO_CXXFLAGS": "",
  "CGO_ENABLED": "1",
  "CGO_LDFLAGS": "",
  "DefaultGODEBUG": "asynctimerchan=1,gotestjsonbuildtext=1,gotypesalias
   =0,httplaxcontentlength=1,httpmuxgo121=1,httpservecontentkeepheaders
   =1,multipathtcp=0,randseednop=0,rsa1024min=0,tls10server=1,tls3des=1,
   tlsmlkem=0,tlsrsakex=1,tlsunsafeekm=1,winreadlinkvolume=0,winsymlink
   =0,x509keypairleaf=0,x509negativeserial=1,x509rsacrt=0,
   x509usepolicies=0",
```

```
  "GOAMD64": "v1",
  "GOARCH": "amd64",
  "GOOS": "linux"
 },
 "inputs": [
  {
   "path": "a.txt",
   "sha256": "
    b6a98d9ce9a2d9149288fa3df42d377c3e42737afdcdaf714e33c0a100b51060",
   "size": 6
  },
  {
   "path": "b.txt",
   "sha256": "
    f2c82decdd7181cf98945929a62598db7e6b477e11f6e0eb0ae97020eff151ad",
   "size": 5
  }
 ]
}
$ golab manifest -check checks.manifest.json papers
manifest matches
$ echo gamma > papers/c.txt; echo changed > papers/b.txt
$ golab manifest -check checks.manifest.json papers
input changed: b.txt
input added: c.txt
golab: 2 differences from the saved manifest
(exit status 1)
$ cut -d, -f1,3,4 golab-ledger.jsonl
{"seq":1,"action":"manifest","subject":"checks"
```

Listing 21.2: Writing a manifest, checking it, and checking it after the inputs change.

The transcript in Listing 21.2 writes the manifest of a directory of two files, prints it, and checks it. The check recomputes the manifest from the present state and prints the differences, and an unchanged state gives the message that the manifest matches. After one file is overwritten and another added, the check names exactly those two inputs and exits with a failure status. The ledger gained one entry, the action `manifest` with the digest, and the check wrote nothing.

Two entries of the printed manifest are worth a closer look. The first is that the build was made with the C compiler enabled. The second is a long setting called `DefaultGODEBUG` that lists compatibility switches. The list is there because the module file declares an older version of the language than the toolchain implements, and the toolchain then makes some of its newer behaviours revert to the old ones. One entry of the list, `httpmuxgo121=1`, makes the standard library's request multiplexer ignore the pattern syntax that later versions added to it. The service does not use that multiplexer, since its routing table is the one of Chapter 17, so the entry changes nothing for the code of this book. A service built on the standard multiplexer under the same module file would route differently from what the version of its toolchain suggests, and the manifest is where that would be found.

21.3 Comparing Two Builds

```
$ golab-static manifest -check checks.manifest.json papers
setting added: -trimpath
setting removed: CGO_CFLAGS
setting removed: CGO_CPPFLAGS
setting removed: CGO_CXXFLAGS
setting CGO_ENABLED: "1" -> "0"
setting removed: CGO_LDFLAGS
golab: 6 differences from the saved manifest
(exit status 1)
```

Listing 21.3: The static binary of Chapter 18 checked against the manifest of the ordinary build.

The binary of the container image was built with the C compiler disabled and with the option that removes file-system paths from the executable. Checked against the manifest made by the ordinary build, it reports the differences in Listing 21.3: the added option, the changed value for the C compiler, and the removal of the settings for the C toolchain's flags, which a build without the C compiler does not record. The two builds were meant to differ and the manifest says how.

The same listing shows a limit. The container build also passed an option to the linker that strips symbol tables, and that option does not appear in the differences. In this toolchain version the settings embedded in the binary do not include it, so a binary built with it and one built without it have equal manifests. A manifest identifies a build only up to what the toolchain chooses to record, and the missing option is a case where two different binaries are indistinguishable by their manifests. The first exercise closes that gap with the digest of the executable itself.

A second limit concerns the source. When the source is not under version control, as it is not in the directory where these transcripts were produced, the settings contain no revision, and two different source trees compiled with the same toolchain and flags yield the same manifest. The record of the revision is only as good as the habit of building from a clean checkout, and when the tree is not clean, the setting that says so is the one a careful reader looks for.

21.4 What the Manifest Does Not Establish

A manifest says what an experiment started from. It does not say that the experiment, run again from the same start, would give the same result. That is a property of the program, and it holds or fails for reasons the manifest cannot see: use of the clock, the order in which goroutines finish, the iteration order of a map, and the contents of the environment all affect the outcome of programs, and a manifest records none of them. The experiment of the next chapter has been arranged so that the property can be checked, since it is built from nothing that varies, and its tests include running the search twice and comparing. A benchmark does not have the property, and its manifest would fix the inputs and leave the numbers

free to differ by the noise of the machine.

The manifest also does not establish that the inputs are the right ones. It records which files were used, by digest, and a digest is the identity of content, as Chapter 3 said, and not a statement that the content is correct. An experiment run on the wrong files has a perfectly good manifest.

Exercises

Exercise 21.1. Add the SHA-256 digest of the running executable to the manifest, found through `os.Executable`. Show that two builds that differ only in the stripping option now have different manifests, and say what the change does to the property that moving the program does not change the manifest.

Exercise 21.2. Build the program inside a version-control checkout and again after modifying a file without committing it. Report the settings that differ and explain how a reader of two manifests could tell the builds apart.

Exercise 21.3. Make the check command ignore the settings and compare only the inputs and parameters. Describe a result that this would let through that the full check catches.

Exercise 21.4. Write the manifest of the benchmark run of Chapter 19, including the machine description from the benchmark header as parameters. Run the benchmark twice and compare the manifests and the results, and state what the manifests establish about the difference between the results.

Chapter 22

What a Summary Discards

22.1 The Question

A ledger grows without bound, and sooner or later someone proposes to replace the old part of it with something smaller. The index of Part IV is one such replacement, and it is safe because the ledger stays where it is and the index can be thrown away. The proposal that follows naturally is to keep the summary and discard the records, and the question this chapter asks is what that would cost. The question is put as an experiment on records and ledgers and has nothing to do with any machine-learning model. It concerns the plainest form of the principle that has run through the book, that a record of what happened and a judgment about it are different things, and that a summary which keeps the judgments someone expected to need may have dropped the distinctions that a later judgment requires.

22.2 Summaries, Decisions, and Availability

Fix a set of ledgers. A *summary* is a function that maps each ledger to a smaller description of it. A *decision* is a question about a ledger whose answer someone may later need: what was the last digest recorded for a file, did a failure follow a success, how many records are there. A decision is *available* from a summary if its answer can be computed from the summary alone.

Proposition 22.1. *Let S be a summary and D a decision on a set $\mathcal{L}$ of ledgers. D is available from S on $\mathcal{L}$, meaning that some function f satisfies $f(S(L)) = D(L)$ for every $L \in \mathcal{L}$, if and only if no two ledgers $L_1, L_2 \in \mathcal{L}$ have $S(L_1) = S(L_2)$ and $D(L_1) \neq D(L_2)$.*

Proof. If f exists and $S(L_1) = S(L_2)$, then $D(L_1) = f(S(L_1)) = f(S(L_2)) = D(L_2)$. Conversely, if no such pair exists, define $f(s)$ for a summary value s in the image of S to be $D(L)$ for any L with $S(L) = s$. The choice does not matter, since all such ledgers agree on D, and so f is well defined and satisfies the equation. □

The proposition turns a question about computation into a question about pairs, and pairs can be searched for. A pair of ledgers with equal summaries and different answers is a *counterexample*, and a single counterexample settles the matter for every set of ledgers

that contains both. The converse is weaker. Finding no counterexample among the ledgers examined shows that the decision is available on those ledgers, and it shows nothing about longer ones unless an argument covers them.

A second fact orders the summaries. Call S' a refinement of S if $S'(L_1) = S'(L_2)$ always implies $S(L_1) = S(L_2)$, which means that S' keeps at least what S keeps. Every decision available from S is then available from S', because a counterexample for S' would be one for S. A summary that adds to another never loses a decision, and the table below shows the relation in the rows for the index and the counts.

22.3 The Search

The package `summary` makes the search concrete. It fixes a small alphabet of seven events: two successful verifications of one file with different digests, one successful verification of a second file, a failed verification of each file, and a run of a program that succeeds or fails. It turns every sequence of events of length zero through six into a real ledger chain in memory, using the function `ledger.Extend`, and so the records are the genuine article, with sequence numbers, predecessor digests, and their own digests. For each ledger it computes six summaries and six decisions, groups the ledgers by summary value, and looks, for each summary and decision, for a group in which the answers differ. Lengths are visited in increasing order, so the counterexample reported for a cell is among the shortest that exist.

```
// Package summary studies what is lost when a ledger is replaced by a smaller
// description of it. A summary is a function of the ledger. A decision is a
// question about the ledger whose answer someone may later need. A decision
// is available from a summary exactly when no two ledgers with the same
// summary give different answers, because only then can the answer be
// computed from the summary alone. The package searches all ledgers up to a
// given length for a pair that breaks this, and reports the shortest pair it
// finds. The results concern records and ledgers; they say nothing about any
// learned model.
package summary

import (
    "encoding/json"
    "fmt"
    "sort"
    "strings"
    "time"
```

```
18
19      "golab/internal/index"
20      "golab/internal/ledger"
21  )
22
23  // Event is one thing that can happen in the small world the search
        explores.
24  type Event struct {
25      Action  string // verify-file, verify-file-failed, or run
26      Subject string // the path, or the program
27      Detail  string // the digest for verify-file, the outcome for run
28  }
29
30  func (e Event) String() string {
31      switch e.Action {
32      case "verify-file":
33          return fmt.Sprintf("verify %s=%s", e.Subject, e.Detail)
34      case "verify-file-failed":
35          return fmt.Sprintf("fail %s", e.Subject)
36      }
37      return "run " + e.Detail
38  }
39
40  // Alphabet is the set of events the search combines. Two paths, two
        digests
41  // for the first, one for the second, a failure for each, and a run
        that
42  // succeeds or fails.
43  var Alphabet = []Event{
44      {"verify-file", "a", "d1"},
45      {"verify-file", "a", "d2"},
46      {"verify-file", "b", "d1"},
47      {"verify-file-failed", "a", ""},
48      {"verify-file-failed", "b", ""},
49      {"run", "prog", "succeeded"},
50      {"run", "prog", "failed"},
51  }
52
53  // Records turns a sequence of events into a real ledger chain, in
        memory.
54  func Records(events []Event) ([]ledger.Record, error) {
55      var recs []ledger.Record
56      t := time.Unix(1_700_000_000, 0)
57      for _, e := range events {
58          var detail map[string]string
59          switch e.Action {
60          case "verify-file":
61              detail = map[string]string{"sha256": e.Detail}
62          case "run":
63              detail = map[string]string{"outcome": e.Detail}
64          }
```

```
        r, err := ledger.Extend(recs, e.Action, e.Subject, detail, t)
        if err != nil {
            return nil, err
        }
        recs = append(recs, r)
    }
    return recs, nil
}

// A Summarizer reduces a ledger to a canonical string. Two ledgers have the
// same summary exactly when the strings are equal.
type Summarizer struct {
    Name string
    Fn   func([]ledger.Record) string
}

// A Decision answers one question about a ledger.
type Decision struct {
    Name string
    Fn   func([]ledger.Record) string
}

func counts(rs []ledger.Record) map[string]int {
    m := map[string]int{"total": len(rs)}
    for _, r := range rs {
        m[r.Action]++
    }
    return m
}

func entries(rs []ledger.Record) []index.Entry {
    x, err := index.Build(rs)
    if err != nil {
        panic(err) // the search builds only valid chains
    }
    es := x.Entries()
    for i := range es {
        es[i].Seq = 0 // the sequence number of the last touch records position, which is not part of the summary
    }
    return es
}

func j(v any) string {
    b, err := json.Marshal(v)
    if err != nil {
        panic(err)
    }
    return string(b)
}
```

```
114
115 // Summarizers returns the summaries compared in Chapter 22.
116 func Summarizers() []Summarizer {
117     return []Summarizer{
118         {"every record", func(rs []ledger.Record) string {
119             var sb strings.Builder
120             for _, r := range rs {
121                 sb.WriteString(r.Digest)
122             }
123             return sb.String()
124         }},
125         {"action counts", func(rs []ledger.Record) string { return j(
    counts(rs)) }},
126         {"per-path index", func(rs []ledger.Record) string { return j
    (entries(rs)) }},
127         {"index and counts", func(rs []ledger.Record) string { return
     j([]any{entries(rs), counts(rs)}) }},
128         {"last three records", func(rs []ledger.Record) string {
129             if len(rs) > 3 {
130                 rs = rs[len(rs)-3:]
131             }
132             var parts []string
133             for _, r := range rs {
134                 parts = append(parts, r.Action+"|"+r.Subject+"|"+r.
    Detail["sha256"]+"|"+r.Detail["outcome"])
135             }
136             return strings.Join(parts, ";")
137         }},
138         {"index and two flags", func(rs []ledger.Record) string {
139             return j([]any{entries(rs), failAfterSuccess(rs, "a"),
    runFailedSinceLastAttempt(rs, "b")})
140         }},
141     }
142 }
143
144 func failAfterSuccess(rs []ledger.Record, path string) bool {
145     seen := false
146     for _, r := range rs {
147         if r.Subject != path {
148             continue
149         }
150         if r.Action == "verify-file" {
151             seen = true
152         } else if r.Action == "verify-file-failed" && seen {
153             return true
154         }
155     }
156     return false
157 }
158
159 func runFailedSinceLastAttempt(rs []ledger.Record, path string) bool
```

```
    {
     last := -1
     for i, r := range rs {
         if r.Subject == path && (r.Action == "verify-file" || r.
    Action == "verify-file-failed") {
             last = i
         }
     }
     for _, r := range rs[last+1:] {
         if r.Action == "run" && r.Detail["outcome"] == "failed" {
             return true
         }
     }
     return false
}

// Decisions returns the questions asked of every summary.
func Decisions() []Decision {
    return []Decision{
        {"last digest of a", func(rs []ledger.Record) string {
            d := "none"
            for _, r := range rs {
                if r.Action == "verify-file" && r.Subject == "a" {
                    d = r.Detail["sha256"]
                }
            }
            return d
        }},
        {"is b verified now", func(rs []ledger.Record) string {
            s := "never"
            for _, r := range rs {
                if r.Subject == "b" {
                    switch r.Action {
                    case "verify-file":
                        s = "yes"
                    case "verify-file-failed":
                        s = "no"
                    }
                }
            }
            return s
        }},
        {"failures of a", func(rs []ledger.Record) string {
            n := 0
            for _, r := range rs {
                if r.Action == "verify-file-failed" && r.Subject == "
    a" {
                    n++
                }
            }
            return fmt.Sprint(n)
```

```
        }},
        {"a failed after a success", func(rs []ledger.Record) string { return fmt.Sprint(failAfterSuccess(rs, "a")) }},
        {"run failed since b's last attempt", func(rs []ledger.Record) string {
            return fmt.Sprint(runFailedSinceLastAttempt(rs, "b"))
        }},
        {"number of records", func(rs []ledger.Record) string { return fmt.Sprint(len(rs)) }},
    }
}

// Pair is two ledgers that have the same summary and give different answers.
type Pair struct {
    A, B       []Event
    AnsA, AnsB string
}

// Cell is the result for one summary and one decision.
type Cell struct {
    Available      bool
    Counterexample *Pair
}

// Result is the whole table.
type Result struct {
    N           int
    Ledgers     int
    Summarizers []string
    Decisions   []string
    Distinct    []int    // number of distinct summaries, per summarizer
    Cells       [][]Cell // [summarizer][decision]
}

// Run examines every sequence of events of length 0 through n. Lengths are
// visited in increasing order, so a counterexample, once found, is among the
// shortest that exist.
func Run(n int) (Result, error) {
    sums, decs := Summarizers(), Decisions()
    res := Result{N: n}
    for _, s := range sums {
        res.Summarizers = append(res.Summarizers, s.Name)
    }
    for _, d := range decs {
        res.Decisions = append(res.Decisions, d.Name)
    }
    type seen struct {
```

```
        ans string
        ev  []Event
    }
    table := make([][]map[string]seen, len(sums)) // [summarizer][
decision] summary -> first answer
    res.Cells = make([][]Cell, len(sums))
    for i := range sums {
        table[i] = make([]map[string]seen, len(decs))
        res.Cells[i] = make([]Cell, len(decs))
        for k := range decs {
            table[i][k] = map[string]seen{}
            res.Cells[i][k].Available = true
        }
    }
    distinct := make([]map[string]bool, len(sums))
    for i := range distinct {
        distinct[i] = map[string]bool{}
    }

    k := len(Alphabet)
    for length := 0; length <= n; length++ {
        idx := make([]int, length)
        for {
            events := make([]Event, length)
            for p, a := range idx {
                events[p] = Alphabet[a]
            }
            recs, err := Records(events)
            if err != nil {
                return Result{}, err
            }
            res.Ledgers++
            answers := make([]string, len(decs))
            for d := range decs {
                answers[d] = decs[d].Fn(recs)
            }
            for i, s := range sums {
                key := s.Fn(recs)
                distinct[i][key] = true
                for d := range decs {
                    prev, ok := table[i][d][key]
                    if !ok {
                        table[i][d][key] = seen{answers[d], events}
                        continue
                    }
                    if prev.ans != answers[d] && res.Cells[i][d].
Available {
                        res.Cells[i][d] = Cell{false, &Pair{prev.ev,
events, prev.ans, answers[d]}}
                    }
                }
```

```
            }
            // advance the counter
            p := length - 1
            for p >= 0 {
                idx[p]++
                if idx[p] < k {
                    break
                }
                idx[p] = 0
                p--
            }
            if p < 0 {
                break
            }
        }
    }
    for i := range distinct {
        res.Distinct = append(res.Distinct, len(distinct[i]))
    }
    return res, nil
}

// Describe writes a sequence of events on one line.
func Describe(es []Event) string {
    if len(es) == 0 {
        return "(empty ledger)"
    }
    parts := make([]string, len(es))
    for i, e := range es {
        parts[i] = e.String()
    }
    return strings.Join(parts, "; ")
}

// Table renders the result as text: one row per summary, one column per
// decision, with "yes" where the decision is available and "no" where a
// counterexample exists, followed by the counterexamples.
func (r Result) Table() string {
    var sb strings.Builder
    fmt.Fprintf(&sb, "%d ledgers of up to %d records\n\n", r.Ledgers, r.N)
    fmt.Fprintf(&sb, "%-21s %8s", "summary", "distinct")
    for i := range r.Decisions {
        fmt.Fprintf(&sb, "%5s", fmt.Sprintf("D%d", i+1))
    }
    sb.WriteString("\n")
    for i, s := range r.Summarizers {
        fmt.Fprintf(&sb, "%-21s %8d", s, r.Distinct[i])
        for d := range r.Decisions {
```

```
            if r.Cells[i][d].Available {
                sb.WriteString("  yes")
            } else {
                sb.WriteString("   no")
            }
        }
        sb.WriteString("\n")
    }
    sb.WriteString("\n")
    for i, d := range r.Decisions {
        fmt.Fprintf(&sb, "D%d  %s\n", i+1, d)
    }
    var shown []string
    for i, s := range r.Summarizers {
        for d := range r.Decisions {
            if c := r.Cells[i][d]; c.Counterexample != nil {
                p := c.Counterexample
                shown = append(shown, fmt.Sprintf("%s, D%d: [%s] and
    [%s] give %q and %q", s, d+1, Describe(p.A), Describe(p.B), p.AnsA
    , p.AnsB))
            }
        }
    }
    sort.Strings(shown)
    sb.WriteString("\nshortest counterexamples found, for summaries
    other than \"every record\":\n")
    // Keep the output readable: show one counterexample per decision
     that fails somewhere,
    // for the most informative summary (the per-path index).
    for _, line := range shown {
        if strings.HasPrefix(line, "per-path index") {
            sb.WriteString("  " + line + "\n")
        }
    }
    return sb.String()
}
```

Listing 22.1: The search for what summaries lose.

The six summaries are the full record, represented by the digests of all its records; the counts of records by action; the per-path index of Chapter 14, without the sequence numbers and without the head digest; the index together with the counts; the last three records; and the index together with two flags that were added after seeing the decisions. The six decisions are the last digest recorded for the first file, whether the second file is currently verified, the number of failures recorded for the first file, whether the first file failed after it had succeeded, whether a run failed since the second file's last verification attempt, and the number of records. The summaries and decisions are chosen by the author and they are a sample, not a census of the questions a reader might ask. The experiment is a demonstration that loss occurs and of what form it takes.

```
$ golab experiment compress -n 6
137257 ledgers of up to 6 records

summary               distinct   D1   D2   D3   D4   D5   D6
every record            137257  yes  yes  yes  yes  yes  yes
action counts               84   no   no   no   no   no  yes
per-path index             611  yes  yes  yes   no   no   no
index and counts          1190  yes  yes  yes   no   no  yes
last three records         400   no   no   no   no   no   no
index and two flags       1049  yes  yes  yes  yes  yes   no

D1  last digest of a
D2  is b verified now
D3  failures of a
D4  a failed after a success
D5  run failed since b's last attempt
D6  number of records

shortest counterexamples found, for summaries other than "every record":
  per-path index, D4: [verify a=d1; fail a; verify a=d1] and [fail a;
    verify a=d1; verify a=d1] give "true" and "false"
  per-path index, D5: [(empty ledger)] and [run failed] give "false" and
     "true"
  per-path index, D6: [(empty ledger)] and [run succeeded] give "0" and
    "1"
```

Listing 22.2: The search over all ledgers of up to six records.

22.4 Reading the Table

The first row of Listing 22.2 is the control. The full record distinguishes all 137,257 ledgers, and every decision is available from it, as it must be. The search found no flaw in the machinery, and a test checks the same thing.

The row for the counts reduces the 137,257 ledgers to 84 distinct values and keeps one decision, the number of records. It cannot say anything about a particular file, because it counts actions and not subjects. The row for the per-path index reduces the ledgers to 611 values and keeps the first three decisions. These are the questions the index was designed to answer, and for them the experiment confirms what the structure of the index already argued: the answers are fields of the entries, so no two ledgers with the same entries can disagree about them. It loses the other three, and the counterexamples say why.

The fourth decision, whether the file failed after it had succeeded, is lost because the entry records how many attempts and failures there were and whether the last attempt succeeded, and not their order. The pair printed for it has three events in each ledger: a success, a failure, and a second success of the same digest in one, and a failure followed by two identical successes in the other. Both leave the file verified with the same digest, two successes, and one failure, and the index cannot tell them apart. In the first the failure came after a success, and in the second it did not. The fifth decision is lost for a simpler reason.

The index takes no notice of runs, so a ledger that holds a failed run and a ledger that holds nothing have the same index. The sixth is lost because the index counts only verification records. The shortest counterexample for it has an empty ledger on one side and a ledger of one successful run on the other.

The row for the index with the counts keeps everything the two parts keep, as the refinement relation requires, and adds nothing that neither had. The row for the last three records shows that recency is not sufficiency. A window of the most recent events keeps no decision at all, since a ledger can have any history before the window. The last row is the instructive one. With two flags added that answer the fourth and fifth decisions, the summary makes both of them available, and the sixth decision, which it did not anticipate and for which it has no count, is lost. The two flags could have been chosen only by knowing the questions in advance. A summary can always be extended to keep the answer to a question that is known, and it cannot be extended to keep the answer to one that has not been asked.

22.5 The Digest That Answers Nothing

One summary has been left out of the table on purpose. Suppose the summary is the number of records together with the digest of the last record. Since the digest depends on every record before it, two different ledgers have different digests unless a collision exists, and the summary is therefore injective, so by Proposition 22.1 every decision is available from it. This is correct and useless. The function f whose existence the proposition promises would have to recover the ledger from its digest, and nobody can compute that function. The proposition says that an answer exists, and does not say that it is computable by any procedure that is cheaper than keeping the ledger.

The observation explains why the head digest in the index is not there to answer questions. It binds the index to one ledger, so that a mismatch is noticed, and it does so without making the index any more informative. It also marks the line between two uses of a summary. A summary that is kept alongside the ledger can be checked against it, and recomputed from it whenever there is doubt, and the doubt is cheap to resolve. A summary that is kept in place of the ledger can be checked against nothing, and the distinctions it dropped cannot be recovered when a later decision needs them.

22.6 What the Experiment Does Not Show

The experiment shows the form of the loss and does not measure its importance. A counterexample of three events establishes that the index cannot answer one question, and says nothing about whether anyone will ever ask it. The alphabet has seven events and two files, and a real ledger has many more, so the table is a small model of the situation. The cells marked *no* are exact: each has a pair that anyone can check, and the test suite checks every pair afresh. The cells marked *yes* for the index and the counts are backed by the argument that the answers are fields of the summary, and for the others by the search over ledgers up to six records, which is evidence for those lengths and not a proof beyond them. The search takes about fifteen seconds on the machine that produced the transcript,

and its cost grows by a factor of seven with each added record, which limits how far the same method can go. The experiment is deterministic, so the manifest of Chapter 21 that records its one parameter is sufficient to reproduce the table.

The practical lesson is the one the ordering of the chapters has prepared. Derived data is safe to throw away when the source stays. A summary that replaces its source is a bet on which questions will matter, and the experiment shows exactly where such a bet can be lost.

Exercises

Exercise 22.1. Add a seventh summary that keeps the index and the last event, and a seventh decision that asks what the final event of the ledger was. Predict the new row and column before running the search, and compare.

Exercise 22.2. Prove that the refinement relation of this chapter is a preorder, and show that the summary of the index with the counts refines both of its parts. Use the proof to explain why the table cannot contain a decision that is available from the index but not from the index with the counts.

Exercise 22.3. Extend the alphabet with a third file and report how the number of ledgers and the running time change. State what the larger alphabet could change about the cells marked *yes*, and what it could not change about the cells marked *no*.

Exercise 22.4. Design a summary of a ledger that is compact, makes the fourth and fifth decisions available, and is built without knowing those two decisions in advance. Say what you gave up to do it, and test your design with the search.

Conclusion

The book began with a counter that could not go below zero and ended with a search over every small ledger for the questions a summary cannot answer. Between them it built one program in six stages, and the same distinction was drawn at each stage: what the compiler guarantees, what a library guarantees under stated assumptions, and what the program merely arranges and checks by tests. The distinction is the book's main claim about the language. Go gives a programmer types that cannot be forged outside their package, interfaces that name only what a consumer needs, errors that are values to be examined, and goroutines that are cheap enough to use freely, and it leaves unenforced almost everything else: that errors are examined, that a map is not shared, that a function is pure, that a callback does not block. Good Go is mostly a matter of arranging the code so that the unenforced things become visible and testable, and of saying in the documentation which ones remain a matter of discipline.

The same distinction governs the analogies to the author's other work. The staging of inscription, validation, admission, projection, and activation, the separation of routing from adjudication, and the observation that a summary can discard what a later gate requires are structural analogies. Each was implemented in a small program and tested, and none of the formal properties of the specifications they resemble were claimed or proved here. Where a proposition appears in the text, it is a proposition about this code under the stated assumptions, and where the assumptions are not enforced, the text says so. Appendix C collects the statements in one table, and its second column is the honest summary of the book: a short list of things the toolchain does for the programmer, and a longer list of things the programmer must keep doing.

Several limits remain open and are named in the text where they arise: the ledger is tamper-evident only relative to a head digest held elsewhere, two processes appending to one file can still fork it, the service speaks plain HTTP, the index file is trusted beyond its position in the ledger, and the manifest identifies a build only as far as the toolchain records it. Each could be closed with more machinery, and each closure would raise a new question of the same shape, which is a fair description of how systems of this kind are built.

Appendix A

Complete Source of the Command

The command-line entry point is reproduced in full. Everything else in the `golab` module appears in the body of the chapters.

```
// Command golab is the laboratory tool developed across the book. It
// discovers files, verifies their content, runs external programs, and
// records each action in a hash-chained ledger. Part IV adds an index that is
// derived from the ledger and can always be rebuilt.
package main

import (
    "context"
    "flag"
    "fmt"
    "io"
    "log/slog"
    "net"
    "net/http"
    "os"
    "os/signal"
    "path/filepath"
    "runtime"
    "sort"
    "strconv"
    "strings"
    "sync/atomic"
    "syscall"
    "time"

    "golab/internal/artifact"
    "golab/internal/atomicfile"
    "golab/internal/index"
    "golab/internal/ledger"
    "golab/internal/manifest"
```

```
    "golab/internal/pipeline"
    "golab/internal/pool"
    "golab/internal/runner"
    "golab/internal/service"
    "golab/internal/summary"
)

const usage = `usage: golab [-ledger file] <command> [arguments]

commands:
  discover <dir>            list files and their kinds
  verify [flags] <dir>      hash every file and record the result
  run <dir> <program> ...   run a program in dir and record the outcome
  admit [flags] <file>      run a file of records through the pipeline
  audit                     check that the ledger chain holds
  index [flags]             write the index derived from the ledger
  status [flags] <dir>      compare files with what the ledger last recorded
  serve [flags]             serve the ledger and admission over HTTP
  manifest [flags] [dir]    record or check what an experiment ran on
  experiment compress [-n]  search for what a summary of the ledger loses

flags for verify:
  -workers n     files to hash at once (default: number of CPUs)
  -each          also write one ledger record per file
  -timeout d     stop starting new files after d

flags for index and status:
  -index file    index file (default golab-index.json)
  -rebuild       (index) ignore any existing index and rebuild it
  -workers n     (status) files to hash at once (default: number of CPUs)

flags for manifest:
  -name s        experiment name (default "experiment")
  -params k=v,.. experiment parameters
  -out file      write the manifest to file and record it in the ledger
  -check file    compare a saved manifest with the current state; exit 1 if different

flags for serve (tokens come from the environment, as GOLAB_TOKENS=name=token:rw,...):
  -addr a        address to listen on (default 127.0.0.1:8080)
  -deny, -scope, -gate   the policy applied to submitted records, as for admit
  -log format    request log on standard error: json, text, or off (default json)
```

```

flags for admit:
  -deny a,b      refuse records written by these authors
  -scope prefix  mark records outside this scope prefix as unscoped
  -gate phrase   hold back records whose body contains the phrase
  -project s     project and activate the records visible in scope s
`

func main() {
    os.Exit(run(os.Args[1:], os.Stdout, os.Stderr))
}

// run is separate from main so that it can be tested and so that main has
// exactly one place where the process exit status is decided.
func run(args []string, stdout, stderr *os.File) int {
    fs := flag.NewFlagSet("golab", flag.ContinueOnError)
    fs.SetOutput(stderr)
    ledgerPath := fs.String("ledger", "golab-ledger.jsonl", "path of the ledger file")
    if err := fs.Parse(args); err != nil {
        return 2
    }
    rest := fs.Args()
    if len(rest) == 0 {
        fmt.Fprint(stderr, usage)
        return 2
    }

    var err error
    switch rest[0] {
    case "discover":
        err = cmdDiscover(rest[1:], stdout)
    case "verify":
        err = cmdVerify(*ledgerPath, rest[1:], stdout)
    case "run":
        err = cmdRun(*ledgerPath, rest[1:], stdout)
    case "admit":
        err = cmdAdmit(*ledgerPath, rest[1:], stdout)
    case "audit":
        err = cmdAudit(*ledgerPath, stdout)
    case "index":
        err = cmdIndex(*ledgerPath, rest[1:], stdout)
    case "status":
        err = cmdStatus(*ledgerPath, rest[1:], stdout)
    case "serve":
        err = cmdServe(*ledgerPath, rest[1:], stdout)
    case "manifest":
        err = cmdManifest(*ledgerPath, rest[1:], stdout)
    case "experiment":
        err = cmdExperiment(rest[1:], stdout)
```

```
    default:
        fmt.Fprintf(stderr, "unknown command %q\n\n%s", rest[0], usage)
        return 2
    }
    if err != nil {
        fmt.Fprintln(stderr, "golab:", err)
        return 1
    }
    return 0
}

func cmdDiscover(args []string, out *os.File) error {
    if len(args) != 1 {
        return fmt.Errorf("discover needs exactly one directory")
    }
    found, err := artifact.Discover(args[0])
    if err != nil {
        return err
    }
    for _, d := range found {
        fmt.Fprintf(out, "%-8s %8d  %s\n", d.Kind, d.Size, d.Path)
    }
    return nil
}

func cmdVerify(ledgerPath string, args []string, out *os.File) error {
    fs := flag.NewFlagSet("verify", flag.ContinueOnError)
    workers := fs.Int("workers", runtime.NumCPU(), "number of files to hash at once")
    each := fs.Bool("each", false, "also write one ledger record per file")
    timeout := fs.Duration("timeout", 0, "stop starting new files after this long (0 means no limit)")
    if err := fs.Parse(args); err != nil {
        return err
    }
    if fs.NArg() != 1 {
        return fmt.Errorf("verify needs exactly one directory")
    }
    dir := fs.Arg(0)

    found, err := artifact.Discover(dir)
    if err != nil {
        return err
    }

    ctx := context.Background()
    if *timeout > 0 {
        var cancel context.CancelFunc
```

```
        ctx, cancel = context.WithTimeout(ctx, *timeout)
        defer cancel()
    }

    w := ledger.NewWriter(ledgerPath, 64)
    defer w.Close()

    opts := pool.Options{Workers: *workers}
    var eachErr atomic.Pointer[error]
    if *each {
        opts.OnResult = func(o pool.Outcome) {
            detail := map[string]string{"kind": string(o.File.Kind)}
            action := "verify-file"
            if o.Err != nil {
                action = "verify-file-failed"
                detail["error"] = o.Err.Error()
            } else {
                detail["sha256"] = o.Verified.SHA256()
            }
            if _, err := w.Append(action, o.File.Path, detail); err
    != nil {
                eachErr.CompareAndSwap(nil, &err)
            }
        }
    }
    outcomes := pool.VerifyAll(ctx, found, opts)

    verified := 0
    var failures []string
    for _, o := range outcomes {
        if o.Err != nil {
            failures = append(failures, o.Err.Error())
            continue
        }
        verified++
        fmt.Fprintf(out, "%s  %s\n", o.Verified.SHA256(), o.Verified.
    Path())
    }
    detail := map[string]string{
        "found":    strconv.Itoa(len(found)),
        "verified": strconv.Itoa(verified),
        "failed":   strconv.Itoa(len(failures)),
        "workers":  strconv.Itoa(*workers),
    }
    if _, err := w.Append("verify", dir, detail); err != nil {
        return err
    }
    if p := eachErr.Load(); p != nil {
        return *p
    }
    if len(failures) > 0 {
```

```
        return fmt.Errorf("%d files could not be verified: %s", len(failures), strings.Join(failures, "; "))
    }
    return nil
}

func cmdRun(ledgerPath string, args []string, out *os.File) error {
    if len(args) < 2 {
        return fmt.Errorf("run needs a directory and a program")
    }
    res, runErr := runner.Run(args[0], time.Minute, args[1], args[2:]...)
    detail := map[string]string{
        "outcome": string(res.Outcome),
        "exit":    strconv.Itoa(res.ExitCode),
        "elapsed": res.Elapsed.Round(time.Millisecond).String(),
    }
    subject := strings.Join(args[1:], " ")
    if _, err := ledger.Append(ledgerPath, "run", subject, detail, time.Now()); err != nil {
        return err
    }
    if runErr != nil {
        return runErr
    }
    fmt.Fprint(out, res.Stdout)
    fmt.Fprintf(out, "[%s, exit %d, %s]\n", res.Outcome, res.ExitCode, res.Elapsed.Round(time.Millisecond))
    if res.Outcome != runner.Succeeded {
        return fmt.Errorf("program %s", res.Outcome)
    }
    return nil
}

func cmdAudit(ledgerPath string, out *os.File) error {
    records, err := ledger.Load(ledgerPath)
    if err != nil {
        return err
    }
    if err := ledger.Check(records); err != nil {
        return err
    }
    fmt.Fprintf(out, "ledger holds: %d records\n", len(records))
    return nil
}

func cmdAdmit(ledgerPath string, args []string, out *os.File) error {
    fs := flag.NewFlagSet("admit", flag.ContinueOnError)
    deny := fs.String("deny", "", "comma-separated authors to refuse")
    scope := fs.String("scope", "", "required scope prefix")
```

```
    gate := fs.String("gate", "", "phrase that sends a record to review")
    project := fs.String("project", "", "scope to project and activate")
    if err := fs.Parse(args); err != nil {
        return err
    }
    if fs.NArg() != 1 {
        return fmt.Errorf("admit needs exactly one records file")
    }
    path := fs.Arg(0)

    f, err := os.Open(path)
    if err != nil {
        return err
    }
    records, err := pipeline.LoadJSONL(f)
    f.Close()
    if err != nil {
        return err
    }

    inscribed := make([]pipeline.Inscribed, len(records))
    for i, r := range records {
        inscribed[i] = pipeline.Inscribe(r)
    }
    valid, err := pipeline.Validate(inscribed)
    if err != nil {
        // A structural failure is recorded too: the attempt happened.
        detail := map[string]string{"stage": "validate", "error": err.Error()}
        if _, lerr := ledger.Append(ledgerPath, "admit-failed", path, detail, time.Now()); lerr != nil {
            return lerr
        }
        return err
    }

    var policies []pipeline.Policy
    if *deny != "" {
        policies = append(policies, pipeline.DenyAuthors(strings.Split(*deny, ",")...))
    }
    if *scope != "" {
        policies = append(policies, pipeline.RequireScope(*scope))
    }
    if *gate != "" {
        policies = append(policies, pipeline.GateContaining(*gate))
    }
    admitted := pipeline.Admit(valid, pipeline.AllOf(policies...))
```

```
307
308     for _, j := range admitted.Judgments() {
309         fmt.Fprintf(out, "%-12s %s  %s\n", j.Disposition, j.Record.ID,
        j.Receipt[:12])
310     }
311
312     counts := admitted.Counts()
313     detail := map[string]string{"records": strconv.Itoa(valid.Len())}
314     names := make([]string, 0, len(counts))
315     for d := range counts {
316         names = append(names, string(d))
317     }
318     sort.Strings(names)
319     for _, n := range names {
320         detail[n] = strconv.Itoa(counts[pipeline.Disposition(n)])
321     }
322
323     if *project != "" {
324         active, err := pipeline.Activate(pipeline.Project(admitted, *
       project))
325         if err != nil {
326             detail["activation"] = err.Error()
327             if _, lerr := ledger.Append(ledgerPath, "admit", path,
       detail, time.Now()); lerr != nil {
328                 return lerr
329             }
330             return err
331         }
332         detail["active"] = strconv.Itoa(len(active.Records()))
333         fmt.Fprintf(out, "active in %q: %d records\n", active.Scope(),
        len(active.Records()))
334     }
335     _, err = ledger.Append(ledgerPath, "admit", path, detail, time.
       Now())
336     return err
337 }
338
339 // openIndex loads the ledger, checks its chain, and returns an index
        that
340 // agrees with it, reporting how the index was obtained.
341 func openIndex(ledgerPath, indexPath string, rebuild bool) (*index.
       Index, index.Source, string, error) {
342     records, err := ledger.Load(ledgerPath)
343     if err != nil {
344         return nil, "", "", err
345     }
346     if rebuild {
347         x, err := index.Build(records)
348         return x, index.Rebuilt, "requested", err
349     }
350     return index.Open(indexPath, records)
```

```
}

func cmdIndex(ledgerPath string, args []string, out *os.File) error {
    fs := flag.NewFlagSet("index", flag.ContinueOnError)
    indexPath := fs.String("index", "golab-index.json", "path of the index file")
    rebuild := fs.Bool("rebuild", false, "ignore any existing index and rebuild it")
    if err := fs.Parse(args); err != nil {
        return err
    }
    if fs.NArg() != 0 {
        return fmt.Errorf("index takes no arguments")
    }
    x, src, reason, err := openIndex(ledgerPath, *indexPath, *rebuild)
    if err != nil {
        return err
    }
    if err := x.Save(*indexPath); err != nil {
        return err
    }
    if reason != "" {
        reason = " (" + reason + ")"
    }
    fmt.Fprintf(out, "index %s%s: %d paths from %d ledger records\n",
     src, reason, x.Len(), x.Records())
    return nil
}

// cmdStatus hashes the files under dir and compares each with the index.
// It reads the ledger and the files and writes nothing.
func cmdStatus(ledgerPath string, args []string, out *os.File) error {
    fs := flag.NewFlagSet("status", flag.ContinueOnError)
    indexPath := fs.String("index", "golab-index.json", "path of the index file")
    workers := fs.Int("workers", runtime.NumCPU(), "number of files to hash at once")
    if err := fs.Parse(args); err != nil {
        return err
    }
    if fs.NArg() != 1 {
        return fmt.Errorf("status needs exactly one directory")
    }
    dir := fs.Arg(0)
    x, _, _, err := openIndex(ledgerPath, *indexPath, false)
    if err != nil {
        return err
    }
```

```
    found, err := artifact.Discover(dir)
    if err != nil {
        return err
    }
    outcomes := pool.VerifyAll(context.Background(), found, pool.Options{Workers: *workers})

    type line struct{ state, path string }
    var lines []line
    seen := map[string]bool{}
    for _, o := range outcomes {
        p := o.File.Path
        seen[p] = true
        e, known := x.Lookup(p)
        switch {
        case o.Err != nil:
            lines = append(lines, line{"unreadable", p})
        case !known:
            lines = append(lines, line{"new", p})
        case e.LastDigest == "":
            lines = append(lines, line{"never-verified", p})
        case e.LastDigest != o.Verified.SHA256():
            lines = append(lines, line{"changed", p})
        case !e.Verified:
            lines = append(lines, line{"last-attempt-failed", p})
        default:
            lines = append(lines, line{"unchanged", p})
        }
    }
    prefix := filepath.Clean(dir) + string(filepath.Separator)
    for _, e := range x.Entries() {
        if strings.HasPrefix(e.Path, prefix) && !seen[e.Path] {
            lines = append(lines, line{"missing", e.Path})
        }
    }
    sort.Slice(lines, func(i, j int) bool { return lines[i].path < lines[j].path })
    differ := 0
    for _, l := range lines {
        fmt.Fprintf(out, "%-19s %s\n", l.state, l.path)
        if l.state != "unchanged" {
            differ++
        }
    }
    if differ > 0 {
        return fmt.Errorf("%d of %d paths differ from the ledger", differ, len(lines))
    }
    return nil
}

```

```
// buildPolicy assembles the policy named by the flags shared by admit and
// serve. With no flags set the combination is empty and admits everything.
func buildPolicy(deny, scope, gate string) pipeline.Policy {
    var policies []pipeline.Policy
    if deny != "" {
        policies = append(policies, pipeline.DenyAuthors(strings.Split(deny, ",")...))
    }
    if scope != "" {
        policies = append(policies, pipeline.RequireScope(scope))
    }
    if gate != "" {
        policies = append(policies, pipeline.GateContaining(gate))
    }
    return pipeline.AllOf(policies...)
}

// cmdServe runs the HTTP service until it receives SIGINT or SIGTERM, then
// stops accepting connections, lets requests in progress finish, and closes
// the ledger writer so that every accepted append is on disk.
func cmdServe(ledgerPath string, args []string, out *os.File) error {
    fs := flag.NewFlagSet("serve", flag.ContinueOnError)
    addr := fs.String("addr", "127.0.0.1:8080", "address to listen on")
    deny := fs.String("deny", "", "comma-separated authors to refuse")
    scope := fs.String("scope", "", "required scope prefix")
    gate := fs.String("gate", "", "phrase that sends a record to review")
    logFormat := fs.String("log", "json", "request log format: json, text, or off")
    if err := fs.Parse(args); err != nil {
        return err
    }
    if fs.NArg() != 0 {
        return fmt.Errorf("serve takes no arguments")
    }
    var logger *slog.Logger
    switch *logFormat {
    case "json":
        logger = slog.New(slog.NewJSONHandler(os.Stderr, nil))
    case "text":
        logger = slog.New(slog.NewTextHandler(os.Stderr, nil))
    case "off":
    default:
        return fmt.Errorf("-log must be json, text, or off")
    }
```

```
    tokens, err := service.ParseTokens(os.Getenv("GOLAB_TOKENS"))
    if err != nil {
        return fmt.Errorf("GOLAB_TOKENS: %w", err)
    }

    ln, err := net.Listen("tcp", *addr)
    if err != nil {
        return err
    }
    w := ledger.NewWriter(ledgerPath, 64)
    svc := service.New(w, buildPolicy(*deny, *scope, *gate), tokens)
    if logger != nil {
        svc.Logger = logger
    }
    srv := &http.Server{
        Handler:           svc,
        ReadHeaderTimeout: 5 * time.Second,
        ReadTimeout:       15 * time.Second,
        WriteTimeout:      30 * time.Second,
        IdleTimeout:       60 * time.Second,
    }

    ctx, stop := signal.NotifyContext(context.Background(), os.
    Interrupt, syscall.SIGTERM)
    defer stop()
    serveErr := make(chan error, 1)
    go func() { serveErr <- srv.Serve(ln) }()
    fmt.Fprintf(out, "listening on %s\n", ln.Addr())

    select {
    case err := <-serveErr:
        w.Close()
        return err
    case <-ctx.Done():
    }
    fmt.Fprintln(out, "shutting down")
    shutdownCtx, cancel := context.WithTimeout(context.Background(),
    10*time.Second)
    defer cancel()
    err = srv.Shutdown(shutdownCtx)
    w.Close()
    if err == nil {
        fmt.Fprintln(out, "stopped")
    }
    return err
}

func parseParams(s string) (map[string]string, error) {
    m := map[string]string{}
    for _, item := range strings.Split(s, ",") {
        if item == "" {
```

```
            continue
        }
        k, v, ok := strings.Cut(item, "=")
        if !ok || k == "" {
            return nil, fmt.Errorf("parameter %q: want key=value",
    item)
        }
        m[k] = v
    }
    return m, nil
}

// cmdManifest collects the manifest of an experiment. With -out it
    writes the
// manifest atomically and records its digest in the ledger. With -
    check it
// compares a saved manifest with the present state and lists the
    differences.
func cmdManifest(ledgerPath string, args []string, out *os.File)
    error {
    fs := flag.NewFlagSet("manifest", flag.ContinueOnError)
    name := fs.String("name", "experiment", "experiment name")
    params := fs.String("params", "", "comma-separated key=value
    parameters")
    outPath := fs.String("out", "", "write the manifest here and
    record it in the ledger")
    check := fs.String("check", "", "compare this saved manifest with
     the present state")
    if err := fs.Parse(args); err != nil {
        return err
    }
    if fs.NArg() > 1 {
        return fmt.Errorf("manifest takes at most one directory")
    }
    dir := fs.Arg(0)

    if *check != "" {
        b, err := os.ReadFile(*check)
        if err != nil {
            return err
        }
        saved, err := manifest.Decode(b)
        if err != nil {
            return err
        }
        now, err := manifest.Collect(saved.Name, saved.Params, dir)
        if err != nil {
            return err
        }
        diffs := manifest.Differences(saved, now)
        for _, d := range diffs {
```

```
            fmt.Fprintln(out, d)
        }
        if len(diffs) > 0 {
            return fmt.Errorf("%d differences from the saved manifest", len(diffs))
        }
        fmt.Fprintln(out, "manifest matches")
        return nil
    }

    p, err := parseParams(*params)
    if err != nil {
        return err
    }
    m, err := manifest.Collect(*name, p, dir)
    if err != nil {
        return err
    }
    b, err := m.Encode()
    if err != nil {
        return err
    }
    digest, err := m.Digest()
    if err != nil {
        return err
    }
    if *outPath == "" {
        fmt.Fprintf(out, "%s\n", b)
        return nil
    }
    if err := atomicfile.WriteBytes(*outPath, 0o644, append(b, '\n')); err != nil {
        return err
    }
    detail := map[string]string{"digest": digest, "inputs": strconv.Itoa(len(m.Inputs)), "file": *outPath}
    if _, err := ledger.Append(ledgerPath, "manifest", m.Name, detail, time.Now()); err != nil {
        return err
    }
    fmt.Fprintf(out, "manifest %s: %d inputs, digest %s\n", m.Name, len(m.Inputs), digest)
    return nil
}

func cmdExperiment(args []string, out io.Writer) error {
    if len(args) == 0 || args[0] != "compress" {
        return fmt.Errorf("experiment: the only experiment is compress")
    }
    fs := flag.NewFlagSet("experiment compress", flag.ContinueOnError
```

```
    )
    n := fs.Int("n", 6, "longest ledger to examine (the search visits
    7^n ledgers)")
    if err := fs.Parse(args[1:]); err != nil {
        return err
    }
    if *n < 0 || *n > 7 {
        return fmt.Errorf("-n must be from 0 to 7")
    }
    res, err := summary.Run(*n)
    if err != nil {
        return err
    }
    fmt.Fprint(out, res.Table())
    return nil
}
```

Listing A.1: The `golab` command.

Appendix B

Building and Running

The module requires a Go toolchain and no other dependency. From the directory that contains `go.mod`, the command `go test ./...` builds every package and runs every test, the command `go vet ./...` applies the standard static checks, and the command `go build -o golab ./cmd/golab` produces the executable used in the transcript of Chapter 6. The programs of Chapter 1 live under `examples` and are each run with `go run` followed by the path of the example directory. The service is started with `golab serve`, which reads its tokens from the environment variable `GOLAB_TOKENS`. The script `deploy/build.sh` builds the static binary and, where a container runtime is available, the image of Chapter 18. The commands `manifest` and `experiment compress` of Part VI need no setup; the second takes about fifteen seconds at its default size. The benchmarks of Chapter 19 run with `go test -run xxx -bench .` from the module root. The tests that exercise subprocesses assume that a POSIX shell named `sh` is on the path, which is the case on Linux and under the Windows Subsystem for Linux.

Appendix C

An Inventory of Guarantees

The chapters of Parts I to VI repeatedly distinguish what the Go language or toolchain guarantees from what a library guarantees under assumptions and from what the program merely arranges by convention and checks by tests. Table C.1 collects the distinctions in one place. The table is deliberately conservative. A property is placed in the first column only if a mechanism outside the author's discipline enforces it, and a property that depends on the callers behaving correctly is listed as a convention however carefully it is documented.

Property	Status	Qualification
Code outside `artifact` cannot build a `Verified` with an invented digest	Compiler	Visibility of unexported fields. The zero value remains available to every caller.
A zero `Verified` is not mistaken for evidence	Convention	Callers must test `IsZero`; nothing forces them to.
External modules cannot import `internal` packages	Toolchain	Enforced by the `go` command.
Import cycles do not exist	Compiler	Checked on every build.
A returned error is examined by the caller	Convention	The compiler accepts a discarded error; a static analyser can flag it.
`Kind` takes only its four declared values	Convention	A conversion from any string compiles.
A policy returns only the five verdicts, never *malformed* or *duplicate*	Compiler	The return type has no such values. Integers outside the range can still be converted to `Verdict`, and are refused at run time.
`Inscribe` and `Validate` receive no policy	Compiler	The signatures have no policy parameter. That nothing else influences them is a matter of reading and testing the code.
Adding a policy to `AllOf` cannot admit more records	Proof and tests	Holds for the code as written, including the fail-closed clause.
A change to an earlier ledger record is detected	Library under assumptions	Requires a collision-resistant hash. Rewriting the last record, truncating the file, or rewriting the whole chain is not detected without an externally held head digest.
A file is unchanged between discovery and verification	Not guaranteed	The size check detects some changes only.
Concurrent goroutines in one process cannot fork the ledger	Proof and tests	Requires that one `Writer` is the only appender to the file. Two processes can still fork the chain, as the experiment of Chapter 12 shows.
The pool returns one outcome per file, in input order	Proof and tests	Cancellation stops new work only; a verification in progress is not interrupted.
The code is free of data races	Detector and argument	The race detector sees only the executions that were run.
A reader of a replaced file sees the whole old or whole new contents	Operating system, with tests	Relies on rename within one filesystem. A crash can leave a stray temporary file, and durability of the rename depends on the directory flush.
A record that was written can be read back	Convention, with a check	`Append` refuses records above `MaxLine`; the limit is not enforced by the type system.
Rebuilding the index gives the same bytes as updating it	Proof and tests	Holds while `Apply` stays free of clocks, files, and map iteration order.
The index file has not been edited	Not guaranteed	Only its position in the ledger is checked; entries are trusted. Decisions never depend on it alone.
Routing depends only on the method and the path	Compiler, with reading	`Match` takes only those two strings. That nothing else enters its decision is read from thirty lines.
A handler runs only for a caller that `Permit` allowed	Proof and tests	Local to the package: it holds while `ServeHTTP` is the only way in. An undeclared requirement is refused.
The ledger has a single writer when the service is used	Convention	Holds inside one process, and only if no command-line process or second container appends to the same file.
A bearer token is confidential in transit	Not guaranteed	The service speaks plain HTTP and must be placed behind transport encryption or on a loopback interface.