PART II

Cloud Native Go Constructs

CHAPTER 3
Go Language Foundations

> A language that doesn't affect the way you think about programming is not worth knowing.[1]
>
> —Alan Perlis, *ACM SIGPLAN Notices (September 1982)*

No programming book would be complete without at least a brief refresher of its language of choice, so here we are!

This chapter will differ slightly from the ones in more introductory level books, however, in that we're assuming that you're at least familiar with common coding paradigms but may or may not be a little rusty with the finer points of Go syntax. As such, this chapter will focus as much on Go's nuances and subtleties as its fundamentals. For a deeper dive into the latter, I recommend either *Introducing Go* by Caleb Doxsey (O'Reilly) or *The Go Programming Language* by Alan A. A. Donovan and Brian W. Kernighan (Addison-Wesley Professional)

If you're relatively new to the language, you'll definitely want to read on. Even if you're somewhat comfortable with Go, you might want to skim this chapter: there will be a gem or two in here for you. If you're a seasoned veteran of the language, you can go ahead and move on to the next chapter (or read it ironically and judge me).

1 Perlis, Alan. *ACM SIGPLAN Notices* 17(9), September 1982, pp. 7–13.

Basic Data Types

Go's basic data types, the fundamental building blocks from which more complex types are constructed, can be divided into three subcategories:

- Booleans that contain only one bit of information—`true` or `false`—representing some logical conclusion or state.
- Numeric types that represent simple—variously sized floating point and signed and unsigned integers—or complex numbers.
- Strings that represent an immutable sequence of Unicode code points.

Booleans

The Boolean data type, representing the two logical truth values, exists in some form[2] in every programming language ever devised. It's represented by the `bool` type, a special 1-bit integer type that has two possible values:

- `true`
- `false`

Go supports all of the typical logical operations:

```
and := true && false
fmt.Println(and)        // "false"

or := true || false
fmt.Println(or)         // "true"

not := !true
fmt.Println(not)        // "false"
```

Curiously, Go doesn't include a logical XOR operator. There *is* a `^` operator, but it's reserved for bitwise XOR operations.

2 Earlier versions of C, C++, and Python lacked a native Boolean type, instead representing them using the integers `0` (for `false`) or `1` (for `true`). Some languages like Perl, Lua, and Tcl still use a similar strategy.

Simple Numbers

Go has a small menagerie of systematically named, floating point, and signed and unsigned integer numbers:

Signed integer
: `int8`, `int16`, `int32`, `int64`

Unsigned integer
: `uint8`, `uint16`, `uint32`, `uint64`

Floating point
: `float32`, `float64`

Systematic naming is nice, but code is written by humans with squishy human brains, so the Go designers provided two lovely conveniences.

First, there are two "machine dependent" types, simply called `int` and `uint`, whose size is determined based on available hardware. These are convenient if the specific size of your numbers isn't critical. Sadly, there's no machine-dependent floating-point number type.

Second, two integer types have mnemonic aliases: `byte`, which is an alias for `uint8`; and `rune`, which is an alias for `uint32`.

For most uses, it generally makes sense to just use `int` and `float64`.

Complex Numbers

Go offers two sizes of *complex numbers*, if you're feeling a little imaginative:[3] `complex64` and `complex128`. These can be expressed as an *imaginary literal* by a floating point immediately followed by an `i`:

```
var x complex64 = 3.1415i
fmt.Println(x)                  // "(0+3.1415i)"
```

Complex numbers are very neat but don't come into play all that often, so I won't drill down into them here. If you're as fascinated by them as I hope you are, *The Go Programming Language* by Donovan and Kernighan gives them the full treatment they deserve.

3 See what I did there?

Strings

A *string* represents a sequence of Unicode code points. Strings in Go are immutable: once created, it's not possible to change a string's contents.

Go supports two styles of string literals, the double-quote style (or interpreted literals) and the back-quote style (or raw string literals). For example, the following two string literals are equivalent:

```
// The interpreted form
"Hello\nworld!\n"

// The raw form
`Hello
world!`
```

In this interpreted string literal, each `\n` character pair will be escaped as one newline character, and each `\"` character pair will be escaped as one double-quote character.

Behind the scenes, a string is actually just a wrapper around a slice of UTF-8 encoded `byte` values, so any operation that can be applied to slices and arrays can also be applied to strings. If you aren't clear on slices yet, you can take this moment to read ahead to "Slices" on page 37.

Variables

Variables can be declared by using the `var` keyword to pair an identifier with some typed value, and may be updated at any time, with the general form:

```
var name type = expression
```

However, there is considerable flexibility in variable declaration:

- With initialization: `var foo int = 42`
- Of multiple variables: `var foo, bar int = 42, 1302`
- With type inference: `var foo = 42`
- Of mixed multiple types: `var b, f, s = true, 2.3, "four"`
- Without initialization (see "Zero Values" on page 33): `var s string`

Go is very opinionated about clutter: it *hates* it. If you declare a variable in a function but don't use it, your program will refuse to compile.

Short Variable Declarations

Go provides a bit of syntactic sugar that allows variables within functions to be simultaneously declared and assigned by using the `:=` operator in place of a `var` declaration with an implicit type.

Short variable declarations have the general form:

```
name := expression
```

These can be used to declare both single and multiple assignments:

- With initialization: `percent := rand.Float64() * 100.0`
- Multiple variables at once: `x, y := 0, 2`

In practice, short variable declarations are the most common way that variables are declared and initialized in Go; `var` is usually only used either for local variables that need an explicit type, or to declare a variable that will be assigned a value later.

Remember that `:=` is a declaration, and `=` is an assignment. A `:=` operator that only attempts to redeclare existing variables will fail at compile time.

Interestingly (and sometimes confusingly), if a short variable declaration has a mix of new and existing variables on its left-hand side, the short variable declaration acts like an assignment to the existing variables.

Zero Values

When a variable is declared without an explicit value, it's assigned to the *zero value* for its type:

- Integers: `0`
- Floats: `0.0`
- Booleans: `false`
- Strings: `""` (the empty string)

To illustrate, let's define four variables of various types, without explicit initialization:

```
var i int
var f float64
var b bool
var s string
```

Now, if we were to use these variables we'd find that they were, in fact, already initialized to their zero values:

```
fmt.Printf("integer: %d\n", i)   // integer: 0
fmt.Printf("float: %f\n", f)     // float: 0.000000
fmt.Printf("boolean: %t\n", b)   // boolean: false
fmt.Printf("string: %q\n", s)    // string: ""
```

You'll notice the use of the `fmt.Printf` function, which allows greater control over output format. If you're not familiar with this function, or with Go's format strings, see the following sidebar.

Formatting I/O in Go

Go's `fmt` package implements several functions for formatting input and output. The most commonly used of these are (probably) `fmt.Printf` and `fmt.Scanf`, which can be used to write to standard output and read from standard input, respectively:

```
func Printf(format string, a ...interface{}) (n int, err error)
func Scanf(format string, a ...interface{}) (n int, err error)
```

You'll notice that each requires a `format` parameter. This is its *format string*: a string embedded with one or more *verbs* that direct how its parameters should be interpreted. For output functions like `fmt.Printf`, the formation of these verbs specifies the format with which the arguments will be printed.

Each function also has a parameter `a`. The `...` (*variadic*) operator indicates that the function accepts zero or more parameters in this place; `interface{}` indicates that the parameter's type is unspecified. Variadic functions will be covered in "Variadic Functions" on page 54; the `interface{}` type in "Interfaces" on page 59.

Some of the common verb flags used in format strings include:

```
%v   The value in a default format
%T   A representation of the type of the value
%%   A literal percent sign; consumes no value
%t   Boolean: the word true or false
%b   Integer: base 2
%d   Integer: base 10
%f   Floating point: decimal point but no exponent, e.g. 123.456
%s   String: the uninterpreted bytes of the string or slice
%q   String: a double-quoted string (safely escaped with Go syntax)
```

If you're familiar with C, you may recognize these as somewhat simplified derivations of the flags used in the `printf` and `scanf` functions. A far more complete listing can be found in Go's documentation for the `fmt` package.

The Blank Identifier

The *blank identifier*, represented by the _ (underscore) operator, acts as an anonymous placeholder. It may be used like any other identifier in a declaration, except it doesn't introduce a binding.

It's most commonly used as a way to selectively ignore unneeded values in an assignment, which can be useful in a language that both supports multiple returns and demands there be no unused variables. For example, if you wanted to handle any potential errors returned by `fmt.Printf`, but don't care about the number of bytes it writes,[4] you could do the following:

```
str := "world"

_, err := fmt.Printf("Hello %s\n", str)
if err != nil {
    // Do something
}
```

The blank identifier can also be used to import a package solely for its side effects:

```
import _ "github.com/lib/pq"
```

Packages imported in this way are loaded and initialized as normal, including triggering any of its `init` functions, but are otherwise ignored and need not be referenced or otherwise directly used.

Constants

Constants are very similar to variables, using the `const` keyword to associate an identifier with some typed value. However, constants differ from variables in some important ways. First, and most obviously, attempting to modify a constant will generate an error at compile time. Second, constants *must* be assigned a value at declaration: they have no zero value.

Both `var` and `const` may be used at both the package and function level, as follows:

```
const language string = "Go"

var favorite bool = true

func main() {
    const text = "Does %s rule? %t!"
    var output = fmt.Sprintf(text, language, favorite)

    fmt.Println(output)   // "Does Go rule? true!"
}
```

4 Why would you?

To demonstrate their behavioral similarity, the previous snippet arbitrarily mixes explicit type definitions with type inference for both the constants and variables.

Finally, the choice of `fmt.Sprintf` is inconsequential to this example, but if you're unclear about Go's format strings you can look back to "Formatting I/O in Go" on page 34.

Container Types: Arrays, Slices, and Maps

Go has three first-class container types that can be used to store collections of element values:

ArrayArray
: A fixed-length sequence of zero or more elements of a particular type.

Slice
: An abstraction around an array that can be resized at runtime.

Map
: An associative data structure that allows distinct keys to be arbitrarily paired with, or "mapped to," values.

As container types, all of these have a `length` property that reflects how many elements are stored in that container. The `len` built-in function can be used to find the length of any array, slice (including strings), or map.

Arrays

In Go, as in most other mainstream languages, an *array* is a fixed-length sequence of zero or more elements of a particular type.

Arrays can be declared by including a length declaration. The zero value of an array is an array of the specified length containing zero-valued elements. Individual array elements are indexed from `0` to `N-1`, and can be accessed using the familiar bracket notation:

```
var a [3]int                    // Zero-value array of type [3]int
fmt.Println(a)                  // "[0 0 0]"
fmt.Println(a[1])               // "0"

a[1] = 42                       // Update second index
fmt.Println(a)                  // "[0 42 0]"
fmt.Println(a[1])               // "42"

i := a[1]
fmt.Println(i)                  // "42"
```

Arrays can be initialized using array literals, as follows:

```
b := [3]int{2, 4, 6}
```

You can also have the compiler count the array elements for you:

```
b := [...]int{2, 4, 6}
```

In both cases, the type of `b` is `[3]int`.

As with all container types, the `len` built-in function can be used to discover the length of an array:

```
fmt.Println(len(b))            // "3"
fmt.Println(b[len(b)-1])       // "6"
```

In practice, arrays aren't actually used directly very often. Instead, it's much more common to use *slices*, an array abstraction type that behaves (for all practical purposes) like a resizable array.

Slices

Slices are a data type in Go that provide a powerful abstraction around a traditional array, such that working with slices looks and feels to the programmer very much like working with arrays. Like arrays, slices provide access to a sequence of elements of a particular type via the familiar bracket notation, indexed from `0` to `N-1`. However, where arrays are fixed-length, slices can be resized at runtime.

As shown in Figure 3-1, a slice is actually a lightweight data structure with three components:

- A pointer to some element of a backing array that represents the first element of the slice (not necessarily the first element of the array)
- A length, representing the number of elements in the slice
- A capacity, which represents the upper value of the length

If not otherwise specified, the capacity value equals the number of elements between the start of the slice and the end of the backing array. The built-in `len` and `cap` functions will provide the length and capacity of a slice, respectively.

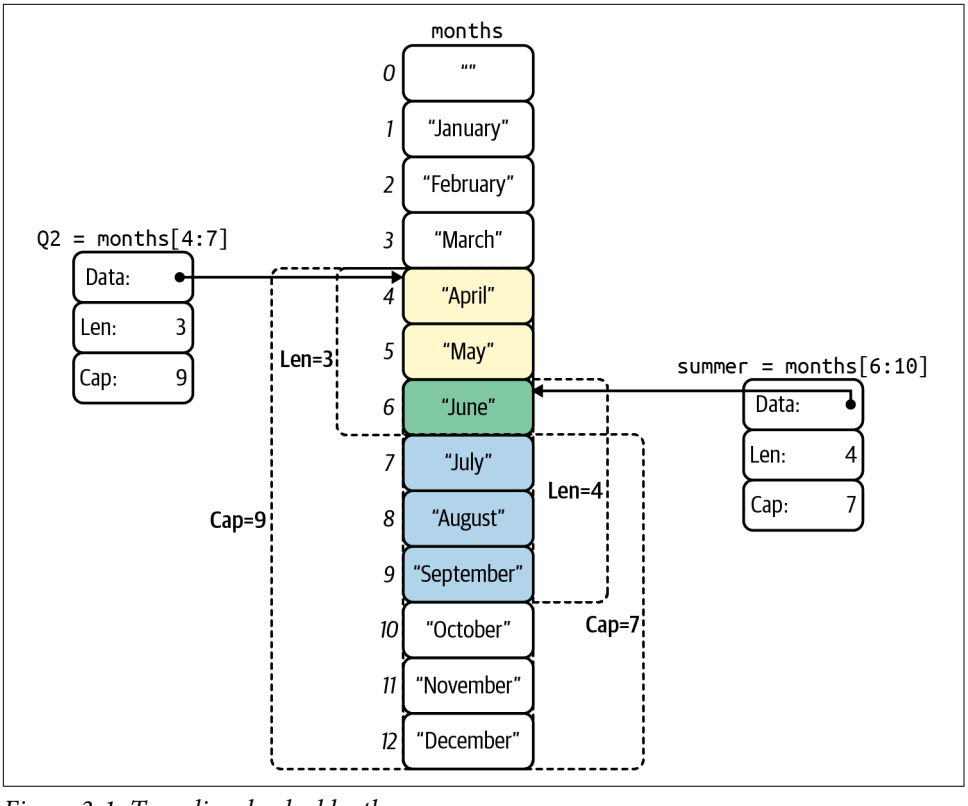

Figure 3-1. Two slices backed by the same array

Working with slices

Creating a slice is somewhat different from creating an array: slices are typed only according to the type of their elements, not their number. The `make` built-in function can be used to create a slice with a nonzero length as follows:

```
n := make([]int, 3)          // Create an int slice with 3 elements

fmt.Println(n)               // "[0 0 0]"
fmt.Println(len(n))          // "3"; len works for slices and arrays

n[0] = 8
n[1] = 16
n[2] = 32

fmt.Println(n)               // "[8 16 32]"
```

As you can see, working with slices feels a lot like working with arrays. Like arrays, the zero value of a slice is a slice of the specified length containing zero-valued

elements, and elements in a slice are indexed and accessed exactly like they are in an array.

A slice literal is declared just like an array literal, except that you omit the element count:

```
m := []int{1}              // A literal []int declaration
fmt.Println(m)             // "[1]"
```

Slices can be extended using the `append` built-in, which returns an extended slice containing one or more new values appended to the original one:

```
m = append(m, 2)           // Append 2 to m
fmt.Println(m)             // "[1 2]"
```

The `append` built-in function also happens to be *variadic*, which means it can accept a variable number of arguments in addition to the slice to be appended. Variadic functions will be covered in more detail in "Variadic Functions" on page 54:

```
m = append(m, 2)           // Append to m from the previous snippet
fmt.Println(m)             // "[1 2]"

m = append(m, 3, 4)
fmt.Println(m)             // "[1 2 3 4]"

m = append(m, m...)        // Append m to itself
fmt.Println(m)             // "[1 2 3 4 1 2 3 4]"
```

Note that the `append` built-in function returns the appended slice rather than modifying the slice in place. The reason for this is that behind the scenes, if the destination has sufficient capacity to accommodate the new elements, then a new slice is constructed from the original underlying array. If not, a new underlying array is automatically allocated.

Note that `append` *returns* the appended slice. Failing to store it is a common error.

The slice operator

Arrays and slices (including strings) support the *slice operator*, which has the syntax `s[i:j]`, where `i` and `j` are in the range `0 ≤ i ≤ j ≤ cap(s)`.

For example:

```
s0 := []int{0, 1, 2, 3, 4, 5, 6}    // A slice literal
fmt.Println(s0)                     // "[0 1 2 3 4 5 6]"
```

In the previous snippet, we define a slice literal. Recall that it closely resembles an array literal, except that it doesn't indicate a size.

If the values of `i` or `j` are omitted from a slice operator, they'll default to `0` and `len(s)`, respectively:

```
s1 := s0[:4]
fmt.Println(s1)                    // "[0 1 2 3]"

s2 := s0[3:]
fmt.Println(s2)                    // "[3 4 5 6]"
```

A slice operator will produce a new slice backed by the same array with a length of `j - i`. Changes made to this slice will be reflected in the underlying array, and subsequently in all slices derived from that same array:

```
s0[3] = 42                         // Change reflected in all 3 slices
fmt.Println(s0)                    // "[0 1 2 42 4 5 6]"
fmt.Println(s1)                    // "[0 1 2 42]"
fmt.Println(s2)                    // "[42 4 5 6]"
```

This effect is illustrated in more detail in Figure 3-1.

Strings as slices

The subject of how Go implements strings under the hood is actually quite a bit more complex than you might expect, involving lots of details like the differences between bytes, characters, and runes; Unicode versus UTF-8 encoding; and the differences between a string and a string literal.

For now it's sufficient to know that Go strings are essentially just read-only slices of bytes that typically (but aren't *required* to) contain a series of UTF-8 sequences representing Unicode code points, called `runes`. Go even allows you to cast your strings into `byte` or `rune` arrays:

```
s := "foö"          // Unicode: f=0x66 o=0x6F ö=0xC3B6
r := []rune(s)
b := []byte(s)
```

By casting the string `s` in this way, we're able to uncover its identity as either a slice of bytes or a slice of runes. We can illustrate this by using `fmt.Printf` with the `%T` (type) and `%v` (value) flags (which we presented in "Formatting I/O in Go" on page 34) to output the results:

```
fmt.Printf("%7T %v\n", s, s)    // "string foö"
fmt.Printf("%7T %v\n", r, r)    // "[]int32 [102 111 246]"
fmt.Printf("%7T %v\n", b, b)    // "[]uint8 [102 111 195 182]"
```

Note that the value of the string literal, `foö`, contains a mix of characters whose encoding can be contained in a single byte (`f` and `o`, encoded as `102` and `111`, respectively) and one character that cannot (`ö`, encoded as `195 182`).

Remember that the `byte` and `rune` types are mnemonic aliases for `uint8` and `int32`, respectively.

Each of these lines print the type and value of the variables passed to it. As expected, the string value, `foö`, is printed literally. The next two lines are interesting, however. The `uint8` (byte) slice contains four bytes, which represent the string's UTF-8 encoding (two 1-byte code points, and one 2-byte code point). The `int32` (rune) slice contains three values that represent the code points of the individual characters.

There's far, far more to string encoding in Go, but we only have so much space. If you're interested in learning more, take a look at Rob Pike's "Strings, Bytes, Runes and Characters in Go" on *The Go Blog* for a deep dive into the subject.

Maps

Go's *map* data type references a *hash table*: an incredibly useful associative data structure that allows distinct keys to be arbitrarily "mapped" to values as key-value pairs. This data structure is common among today's mainstream languages: if you're coming to Go from one of these then you probably already use them, perhaps in the form of Python's `dict`, Ruby's `Hash`, or Java's `HashMap`.

Map types in Go are written `map[K]V`, where `K` and `V` are the types of its keys and values, respectively. Any type that is comparable using the `==` operator may be used as a key, and `K` and `V` need not be of the same type. For example, `string` keys may be mapped to `float32` values.

A map can be initialized using the built-in `make` function, and its values can be referenced using the usual `name[key]` syntax. Our old friend `len` will return the number of key/value pairs in a map; the `delete` built-in can remove key/value pairs:

```
freezing := make(map[string]float32)    // Empty map of string to float32

freezing["celsius"] = 0.0
freezing["fahrenheit"] = 32.0
freezing["kelvin"] = 273.2

fmt.Println(freezing["kelvin"])         // "273.2"
fmt.Println(len(freezing))              // "3"
```

```
delete(freezing, "kelvin")              // Delete "kelvin"
fmt.Println(len(freezing))              // "2"
```

Maps may also be initialized and populated as *map literals*:

```
freezing := map[string]float32{
    "celsius":    0.0,
    "fahrenheit": 32.0,
    "kelvin":     273.2,                // The trailing comma is required!
}
```

Note the trailing comma on the last line. This is not optional: the code will refuse to compile if it's missing.

Map membership testing

Requesting the value of a key that's not present in a map won't cause an exception to be thrown (those don't exist in Go anyway) or return some kind of `null` value. Rather, it returns the zero value for the map's value type:

```
foo := freezing["no-such-key"]          // Get non-existent key
fmt.Println(foo)                        // "0" (float32 zero value)
```

This can be a very useful feature because it reduces a lot of boilerplate membership testing when working with maps, but it can be a little tricky when your map happens to actually contain zero-valued values. Fortunately, accessing a map can also return a second optional `bool` that indicates whether the key is present in the map:

```
newton, ok := freezing["newton"]        // What about the Newton scale?
fmt.Println(newton)                     // "0"
fmt.Println(ok)                         // "false"
```

In this snippet, the value of `newton` is `0.0`. But is that really the correct value,[5] or was there just no matching key? Fortunately, since `ok` is also `false`, we know the latter to be the case.

Pointers

Okay. Pointers. The bane and undoing of undergraduates the world over. If you're coming from a dynamically typed language, the idea of the pointer may seem alien to you. While we're not going to drill down *too* deeply into the subject, we'll do our best to cover it well enough to provide some clarity on the subject.

Going back to first principles, a "variable" is a piece of storage in memory that contains some value. Typically, when you refer to a variable by its name (`foo = 10`) or by

5 In fact, the freezing point of water on the Newton scale actually is 0.0, but that's not important.

an expression (`s[i] = "foo"`), you're directly reading or updating the value of the variable.

A *pointer* stores the *address* of a variable: the location in memory where the value is stored. Every variable has an address, and using pointers allows us to indirectly read or update the value of their variables (illustrated in Figure 3-2):

Retrieving the address of a variable
: The address of a named variable can be retrieved by using the `&` operator. For example, the expression `p := &a` will obtain the address of `a` and assign it to `p`.

Pointer types
: The variable `p`, which you can say "points to" `a`, has a type of `*int`, where the `*` indicates that it's a pointer type that points to an `int`.

Dereferencing a pointer
: To retrieve the value of the value `a` from `p`, you can *dereference* it using a `*` before the pointer variable name, allowing us to indirectly read or update `a`.

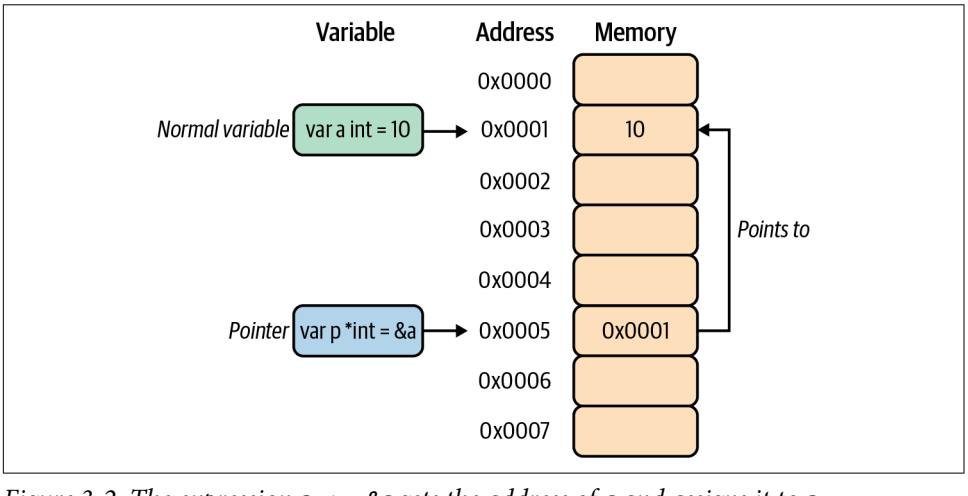

Figure 3-2. The expression `p := &a` gets the address of `a` and assigns it to `p`

Now, to put everything in one place, take a look at the following:

```
var a int = 10

var p *int = &a         // p of type *int points to a
fmt.Println(p)          // "0x0001"
fmt.Println(*p)         // "10"

*p = 20                 // indirectly update a
fmt.Println(a)          // "20"
```

Pointers can be declared like any other variable, with a zero value of `nil` if not explicitly initialized. They're also comparable, being equal only if they contain the same address (that is, they point to the same variable) or if they are both `nil`:

```
var n *int
var x, y int

fmt.Println(n)              // "<nil>"
fmt.Println(n == nil)       // "true" (n is nil)

fmt.Println(x == y)         // "true" (x and y are both zero)
fmt.Println(&x == &x)       // "true" (*x is equal to itself)
fmt.Println(&x == &y)       // "false" (different vars)
fmt.Println(&x == nil)      // "false" (*x is not nil)
```

Because `n` is never initialized, its value is `nil`, and comparing it to `nil` returns `true`. The integers `x` and `y` both have a value of `0`, so comparing their values yields `true`, but they are still distinct variables, and comparing pointers to them still evaluates to `false`.

Control Structures

Any programmer coming to Go from another language will find its suite of control structures to be generally familiar, even comfortable (at first) for those coming from a language heavily influenced by C. However, there are some pretty important deviations in their implementation and usages that might seem odd at first.

For example, control structure statements don't require lots of parentheses. Okay. Less clutter. That's fine.

There's also only one loop type. There is no `while`; only `for`. Seriously! It's actually pretty cool, though. Read on, and you'll see what I mean.

Fun with for

The `for` statement is Go's one and only loop construct, and while there's no explicit `while` loop, Go's `for` can provide all of its functionality, effectively unifying all of the entry control loop types to which you've become accustomed.

Go has no `do-while` equivalent.

The general for statement

The general form of `for` loops in Go is nearly identical to that of other C-family languages, in which three statements—the init statement, the continuation condition, and the post statement—are separated by semicolons in the traditional style. Any variables declared in the init statement will be scoped only to the `for` statement:

```
sum := 0

for i := 0; i < 10; i++ {
    sum += 1
}

fmt.Println(sum)        // "10"
```

In this example, `i` is initialized to 0. At the end of each iteration `i` is incremented by 1, and if it's still less than 10, the process repeats.

Unlike most C-family languages, `for` statements don't require parentheses around their clauses, and braces are required.

In a break from traditional C-style languages, Go's `for` statement's init and post statements are entirely optional. As shown in the code that follows, this makes it considerably more flexible:

```
sum, i := 0, 0

for i < 10 {            // Equivalent to: for ; i < 10;
    sum += i
    i++
}

fmt.Println(i, sum)     // "10 45"
```

The `for` statement in the previous example has no init or post statements, only a bare condition. This is actually a big deal, because it means that `for` is able to fill the role traditionally occupied by the `while` loop.

Finally, omitting all three clauses from a `for` statement creates a block that loops infinitely, just like a traditional `while (true)`:

```
fmt.Println("For ever...")

for {
    fmt.Println("...and ever")
}
```

Because it lacks any terminating condition, the loop in the previous snippet will iterate forever. On purpose.

Looping over arrays and slices

Go provides a useful keyword, `range`, that simplifies looping over a variety of data types.

In the case of arrays and slices, `range` can be used with a `for` statement to retrieve the index and the value of each element as it iterates:

```
s := []int{2, 4, 8, 16, 32}     // A slice of ints

for i, v := range s {           // range gets each index/value
    fmt.Println(i, "->", v)     // Output index and its value
}
```

In the previous example, the values of `i` and `v` will update each iteration to contain the index and value, respectively, of each element in the slice `s`. So the output will look something like the following:

```
0 -> 2
1 -> 4
2 -> 8
3 -> 16
4 -> 32
```

But what if you don't need both of these values? After all, the Go compiler will demand that you use them if you declare them. Fortunately, as elsewhere in Go, the unneeded values can be discarded by using the "blank identifier," signified by the underscore operator:

```
a := []int{0, 2, 4, 6, 8}
sum := 0

for _, v := range a {
    sum += v
}

fmt.Println(sum)    // "20"
```

As in the last example, the value `v` will update each iteration to contain the value of each element in the slice `a`. This time, however, the index value is conveniently ignored and discarded, and the Go compiler stays content.

Looping over maps

The `range` keyword may be also be used with a `for` statement to loop over maps, with each iteration returning the current key and value:

```
m := map[int]string{
    1: "January",
    2: "February",
    3: "March",
    4: "April",
}

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

Note that Go maps aren't ordered, so the output won't be either:

```
3 -> March
4 -> April
1 -> January
2 -> February
```

The if Statement

The typical application of the `if` statement in Go is consistent with other C-style languages, except for the lack of parentheses around the clause and the fact that braces are required:

```
if 7 % 2 == 0 {
    fmt.Println("7 is even")
} else {
    fmt.Println("7 is odd")
}
```

Unlike most C-family languages, `if` statements don't require parentheses around their clauses, and braces are required.

Interestingly, Go allows an initialization statement to precede the condition clause in an `if` statement, allowing for a particularly useful idiom. For example:

```
if _, err := os.Open("foo.ext"); err != nil {
    fmt.Println(err)
} else {
    fmt.Println("All is fine.")
}
```

Note how the `err` variable is being initialized prior to a check for its definition, making it somewhat similar to the following:

```
_, err := os.Open("foo.go")
if err != nil {
    fmt.Println(err)
} else {
    fmt.Println("All is fine.")
}
```

The two constructs aren't exactly equivalent however: in the first example `err` is scoped only to the `if` statement; in the second example `err` is visible to the entire containing function.

The switch Statement

As in other languages, Go provides a `switch` statement that provides a way to more concisely express a series of `if-then-else` conditionals. However, it differs from the traditional implementation in a number of ways that make it considerably more flexible.

Perhaps the most obvious difference to folks coming from C-family languages is that there's no fallthrough between the cases by default; this behavior can be explicitly added by using the `fallthrough` keyword:

```
i := 0

switch i % 3 {
case 0:
    fmt.Println("Zero")
    fallthrough
case 1:
    fmt.Println("One")
case 2:
    fmt.Println("Two")
default:
    fmt.Println("Huh?")
}
```

In this example, the value of `i % 3` is `0`, which matches the first case, causing it to output to word `Zero`. In Go, `switch` cases don't fall through by default, but the existence of an explicit `fallthrough` statement means that the subsequent case is also executed and `One` is printed. Finally, the absence of a `fallthrough` on that case causes the resolution of the `switch` to complete. All told, the following is printed:

```
Zero
One
```

Switches in Go have two interesting properties. First, `case` expressions don't need to be integers, or even constants: the cases will be evaluated from top to bottom, running the first case whose value is equal to the condition expression. Second, if the switch expression is left empty it'll be interpreted as `true`, and will match the first case whose guarding condition evaluates to `true`. Both of these properties are demonstrated in the following example:

```
hour := time.Now().Hour()

switch {
case hour >= 5 && hour < 9:
    fmt.Println("I'm writing")
case hour >= 9 && hour < 18:
    fmt.Println("I'm working")
default:
```

```
        fmt.Println("I'm sleeping")
    }
```

The `switch` has no condition, so it's exactly equivalent to using `switch true`. As such, it matches the first statement whose condition also evaluates to `true`. In my case, `hour` is 23, so the output is "I'm sleeping."[6]

Finally, just as with `if`, a statement can precede the condition expression of a `switch`, in which case any defined values are scoped to the `switch`. For example, the previous example can be rewritten as follows:

```
switch hour := time.Now().Hour(); {  // Empty expression means "true"
case hour >= 5 && hour < 9:
    fmt.Println("I'm writing")
case hour >= 9 && hour < 18:
    fmt.Println("I'm working")
default:
    fmt.Println("I'm sleeping")
}
```

Note the trailing semicolon: this empty expression implies `true`, so that this expression is equivalent to `switch hour := time.Now().Hour(); true` and matches the first true case condition.

Error Handling

Errors in Go are treated as just another value, represented by the built-in `error` type. This makes error handling straightforward: idiomatic Go functions may include an `error`-typed value in its list of returns, which if not `nil` indicates an error state that may be handled via the primary execution path. For example, the `os.Open` function returns a non-`nil` error value when it fails to open a file:

```
file, err := os.Open("somefile.ext")
if err != nil {
    log.Fatal(err)
    return err
}
```

The actual implementation of the `error` type is actually incredibly simple: it's just a universally visible interface that declares a single method:

```
type error interface {
    Error() string
}
```

6 Clearly this code needs to be recalibrated.

This is very different from the exceptions that are used in many languages, which necessitate a dedicated system for exception catching and handling that can lead to confusing and unintuitive flow control.

Creating an Error

There are two simple ways to create error values, and a more complicated way. The simple ways are to use either the `errors.New` or `fmt.Errorf` functions; the latter is handy because it provides string formatting too:

```
e1 := errors.New("error 42")
e2 := fmt.Errorf("error %d", 42)
```

However, the fact that `error` is an interface allows you to implement your own error types, if you need to. For example, a common pattern is to allow errors to be nested within other errors:

```
type NestedError struct {
    Message string
    Err     error
}

func (e *NestedError) Error() string {
    return fmt.Sprintf("%s\n  contains: %s", e.Message, e.Err.Error())
}
```

For more information about errors, and some good advice on error handling in Go, take a look at Andrew Gerrand's "Error Handling and Go" on *The Go Blog*.

Putting the Fun in Functions: Variadics and Closures

Functions in Go work a lot like they do in other languages: they receive parameters, do some work, and (optionally) return something.

But Go functions are built for a level of flexibility not found in many mainstream languages, and can also do a lot of things that many other languages can't, such as returning or accepting multiple values, or being used as first-class types or anonymous functions.

Functions

Declaring a function in Go is similar to most other languages: they have a name, a list of typed parameters, an optional list of return types, and a body. However, Go function declaration differs somewhat from other C-family languages, in that it uses a dedicated `func` keyword; the type for each parameter follows its name; and return types are placed at the end of the function definition header and may be omitted entirely (there's no `void` type).

A function with a return type list must end with a `return` statement, except when execution can't reach the end of the function due to the presence of an infinite loop or a terminal `panic` before the function exits:

```
func add(x int, y int) int {
    return x + y
}

func main() {
    sum := add(10, 5)
    fmt.Println(sum)        // "15"
}
```

Additionally, a bit of syntactic sugar allows the type for a sequence of parameters or returns of the same type to be written only once. For example, the following definitions of `func foo` are equivalent:

```
func foo(i int, j int, a string, b string) { /* ... */ }
func foo(i, j int, a, b string)            { /* ... */ }
```

Multiple return values

Functions can return any number of values. For example, the following `swap` function accepts two strings, and returns two strings. The list of return types for multiple returns must be enclosed in parentheses:

```
func swap(x, y string) (string, string) {
    return y, x
}
```

To accept multiple values from a function with multiple returns, you can use multiple assignment:

```
a, b := swap("foo", "bar")
```

When run, the value of `a` will be "bar," and `b` will be "foo."

Recursion

Go allows *recursive* function calls, in which functions call themselves. Used properly, recursion can be a very powerful tool that can be applied to many types of problems. The canonical example is the calculation of the factorial of a positive integer, the product of all positive integers less than or equal to `n`:

```
func factorial(n int) int {
    if n < 1 {
        return 1
    }
    return n * factorial(n-1)
}

func main() {
```

```
    fmt.Println(factorial(11))      // "39916800"
}
```

For any integer `n` greater than one, `factorial` will call itself with a parameter of `n - 1`. This can add up very quickly!

Defer

Go's `defer` keyword can be used to schedule the execution of a function call for immediately before the surrounding function returns, and is commonly used to guarantee that resources are released or otherwise cleaned up.

For example, to defer printing the text "cruel world" to the end of a function call, we insert the `defer` keyword immediately before it:

```
func main() {
    defer fmt.Println("cruel world")

    fmt.Println("goodbye")
}
```

When the previous snippet is run, it produces the following output, with the deferred output printed last:

```
goodbye
cruel world
```

For a less trivial example, we'll create an empty file and attempt to write to it. A `closeFile` function is provided to close the file when we're done with it. However, if we simply call it at the end of `main`, an error could result in `closeFile` never being called and the file being left in an open state. Therefore, we use a `defer` to ensure that the `closeFile` function is called before the function returns, however it returns:

```
func main() {
    file, err := os.Create("/tmp/foo.txt")  // Create an empty file
    defer closeFile(file)                   // Ensure closeFile(file) is called
    if err != nil {
        return
    }

    _, err = fmt.Fprintln(file, "Your mother was a hamster")
    if err != nil {
        return
    }

    fmt.Println("File written to successfully")
}

func closeFile(f *os.File) {
    if err := f.Close(); err != nil {
        fmt.Println("Error closing file:", err.Error())
```

```
    } else {
        fmt.Println("File closed successfully")
    }
}
```

When you run this code, you should get the following output:

```
File written to successfully
File closed successfully
```

If multiple defer calls are used in a function, each is pushed onto a stack. When the surrounding function returns, the deferred calls are executed in last-in-first-out order. For example:

```
func main() {
    defer fmt.Println("world")
    defer fmt.Println("cruel")
    defer fmt.Println("goodbye")
}
```

This function, when run, will output the following:

```
goodbye
cruel
world
```

Defers are a very useful feature for ensuring that resources are cleaned up. If you're working with external resources, you'll want to make liberal use of them.

Pointers as parameters

Much of the power of pointers becomes evident when they're combined with functions. Typically, function parameters are *passed by value*: when a function is called it receives a copy of each parameter, and changes made to the copy by the function don't affect the caller. However, pointers contain a *reference* to a value, rather than the value itself, and can be used by a receiving function to indirectly modify the value passed to the function in a way that can affect the function caller.

The follow function demonstrates both scenarios:

```
func main() {
    x := 5

    zeroByValue(x)
    fmt.Println(x)              // "5"

    zeroByReference(&x)
    fmt.Println(x)              // "0"
}

func zeroByValue(x int) {
    x = 0
}
```

```
func zeroByReference(x *int) {
    *x = 0                      // Dereference x and set it to 0
}
```

This behavior isn't unique to pointers. In fact, under the hood, several data types are actually references to memory locations, including slices, maps, functions, and channels. Changes made to such *reference types* in a function can affect the caller, without needing to explicitly dereference them:

```
func update(m map[string]int) {
    m["c"] = 2
}

func main() {
    m := map[string]int{ "a" : 0, "b" : 1}

    fmt.Println(m)                  // "map[a:0 b:1]"

    update(m)

    fmt.Println(m)                  // "map[a:0 b:1 c:2]"
}
```

In this example, the map `m` has a length of two when it's passed to the `update` function, which adds the pair `{ "c" : 2 }`. Because `m` is a reference type, it's passed to `update` as a reference to an underlying data structure instead of a copy of one, so the insertion is reflected in `m` in `main` after the `update` function returns.

Variadic Functions

A *variadic function* is one that may be called with zero or more trailing arguments. The most familiar example is the members of the `fmt.Printf` family of functions, which accept a single format specifier string and an arbitrary number of additional arguments.

This is the signature for the standard `fmt.Printf` function:

```
func Printf(format string, a ...interface{}) (n int, err error)
```

Note that it accepts a string, and zero or more `interface{}` values. If you're rusty on the `interface{}` syntax, we'll review it in "Interfaces" on page 59, but you can interpret `interface{}` to mean "some arbitrarily typed thing." What's most interesting here, however, is that the final argument contains an ellipsis (`...`). This is the *variadic operator*, which indicates that the function may be called with any number of arguments of this type. For example, you can call `fmt.Printf` with a format and two differently typed parameters:

```
const name, age = "Kim", 22
fmt.Printf("%s is %d years old.\n", name, age)
```

Within the variadic function, the variadic argument is a slice of the argument type. In the following example, the variadic `factors` parameter of the `product` method is of type `[]int` and may be ranged over accordingly:

```
func product(factors ...int) int {
    p := 1

    for _, n := range factors {
        p *= n
    }

    return p
}

func main() {
    fmt.Println(product(2, 2, 2))   // "8"
}
```

In this example, the call to `product` from `main` uses three parameters (though it could use any number of parameters it likes). In the `product` function, these are translated into an `[]int` slice with the value `{2, 2, 2}` that are iteratively multiplied to construct the final return value of `8`.

Passing slices as variadic values

What if your value is already in slice form, and you still want to pass it to a variadic function? Do you need to split it into multiple individual parameters? Goodness no.

In this case, you can apply the variadic operator after the variable name when calling the variadic function:

```
m := []int{3, 3, 3}
fmt.Println(product(m...))   // "27"
```

Here, you have a variable `m` with the type `[]int`, which you want to pass to the variadic function `product`. Using the variadic operator when calling `product(m...)` makes this possible.

Anonymous Functions and Closures

In Go, functions are *first-class values* that can be operated upon in the same way as any other entity in the language: they have types, may be assigned to variables, and may even be passed to and returned by other functions.

The zero value of a function type is `nil`; calling a `nil` function value will cause a panic:

```
func sum(x, y int) int     { return x + y }
func product(x, y int) int { return x * y }
```

```
func main() {
    var f func(int, int) int    // Function variables have types

    f = sum
    fmt.Println(f(3, 5))        // "8"

    f = product                 // Legal: product has same type as sum
    fmt.Println(f(3, 5))        // "15"
}
```

Functions may be created within other functions as *anonymous functions*, which may be called, passed, or otherwise treated like any other functions. A particularly powerful feature of Go is that anonymous functions have access to the state of their parent, and retain that access *even after* the parent function has executed. This is, in fact, the definition of a *closure*.

A *closure* is a nested function that has access to the variables of its parent function, even after the parent has executed.

Take, for example, the following `incrementor` function. This function has state, in the form of the variable `i`, and returns an anonymous function that increments that value before returning it. The returned function can be said to *close over* the variable `i`, making it a true (if trivial) closure:

```
func incrementer() func() int {
    i := 0

    return func() int {    // Return an anonymous function
        i++                // "Closes over" parent function's i
        return i
    }
}
```

When we call `incrementor`, it creates its own new, local value of `i`, and returns a new anonymous function of type that will increment that value. Subsequent calls to `incrementor` will each receive their own copy of `i`. We can demonstrate that in the following:

```
func main() {
    increment := incrementer()
    fmt.Println(increment())       // "1"
    fmt.Println(increment())       // "2"
    fmt.Println(increment())       // "3"

    newIncrement := incrementer()
    fmt.Println(newIncrement())    // "1"
}
```

As you can see, the `incrementer` provides a new function `increment`; each call to `increment` increments its internal counter by one. When `incrementer` is called again, though, it creates and returns an entirely new function, with its own brand new counter. Neither of these functions can influence the other.

Structs, Methods, and Interfaces

One of the biggest mental switches that people sometimes have to make when first coming to the Go language is that Go isn't a traditional object-oriented language. Not really. Sure, Go has types with methods, which kind of look like objects, but they don't have a prescribed inheritance hierarchy. Instead Go allows components to be assembled into a whole using *composition*.

For example, where a more strictly object-oriented language might have a `Car` class that extends an abstract `Vehicle` class; perhaps it would implement `Wheels` and `Engine`. This sounds fine in theory, but these relationships can grow to become convoluted and hard to manage.

Go's composition approach, on the other hand, allows components to be "put together" without having to define their ontological relationships. Extending the previous example, Go could have a `Car` struct, which could be have its various parts, such as `Wheels` and `Engine`, embedded within it. Furthermore, methods in Go can be defined for any sort of data; they're not just for structs anymore.

Structs

In Go, a *struct* is nothing more than an aggregation of zero or more fields as a single entity, where each field is a named value of an arbitrary type. A struct can be defined using the following `type Name struct` syntax. A struct is never `nil`: rather, the zero value of a struct is the zero value of all of its fields:

```
type Vertex struct {
    X, Y float64
}

func main() {
    var v Vertex            // Structs are never nil
    fmt.Println(v)          // "{0 0}"

    v = Vertex{}            // Explicitly define an empty struct
    fmt.Println(v)          // "{0 0}"

    v = Vertex{1.0, 2.0}    // Defining fields, in order
    fmt.Println(v)          // "{1 2}"

    v = Vertex{Y:2.5}       // Defining specific fields, by label
```

```
    fmt.Println(v)          // "{0 2.5}"
}
```

Struct fields can be accessed using the standard dot notation:

```
func main() {
    v := Vertex{X: 1.0, Y: 3.0}
    fmt.Println(v)                  // "{1 3}"

    v.X *= 1.5
    v.Y *= 2.5

    fmt.Println(v)                  // "{1.5 7.5}"
}
```

Structs are commonly created and manipulated by reference, so Go provides a little bit of syntactic sugar: members of structs can be accessed from a pointer to the struct using dot notation; the pointers are automatically dereferenced:

```
func main() {
    var v *Vertex = &Vertex{1, 3}
    fmt.Println(v)                  // &{1 3}

    v.X, v.Y = v.Y, v.X
    fmt.Println(v)                  // &{3 1}
}
```

In this example, `v` is a pointer to a `Vertex` whose `X` and `Y` member values you want to swap. If you had to dereference the pointer to do this, you'd have to do something like `(*v).X, (*v).Y = (*v).Y, (*v).X`, which is clearly terrible. Instead, automatic pointer dereferencing lets you do `v.X, v.Y = v.Y, v.X`, which is far less terrible.

Methods

In Go, *methods* are functions that are attached to types, including but not limited to structs. The declaration syntax for a method is very similar to that of a function, except that it includes an extra *receiver argument* before the function name that specifies the type that the method is attached to. When the method is called, the instance is accessible by the name specified in the receiver.

For example, our earlier `Vertex` type can be extended by attaching a `Square` method with a receiver named `v` of type `*Vertex`:

```
func (v *Vertex) Square() {    // Attach method to the *Vertex type
    v.X *= v.X
    v.Y *= v.Y
}

func main() {
    vert := &Vertex{3, 4}
    fmt.Println(vert)          // "&{3 4}"

    vert.Square()
    fmt.Println(vert)          // "&{9 16}"
}
```

Receivers are type specific: methods attached to a pointer type can only be called on a pointer to that type.

In addition to structs, you can also claim standard composite types—structs, slices, or maps—as your own, and attach methods to them. For example, we declare a new type, `MyMap`, which is just a standard `map[string]int`, and attach a `Length` method to it:

```
type MyMap map[string]int

func (m MyMap) Length() int {
    return len(m)
}

func main() {
    mm := MyMap{"A":1, "B": 2}

    fmt.Println(mm)             // "map[A:1 B:2]"
    fmt.Println(mm["A"])        // "1"
    fmt.Println(mm.Length())    // "2"
}
```

The result is a new type, `MyMap`, which is (and can be used as) a map of strings to integers, `map[string]int`, but which also has a `Length` method that returns the map's length.

Interfaces

In Go, an *interface* is just a set of method signatures. As in other languages with a concept of an interface, they are used to describe the general behaviors of other types without being coupled to implementation details. An interface can thus be viewed as a *contract* that a type may satisfy, opening the door to powerful abstraction techniques.

For example, a `Shape` interface can be defined that includes an `Area` method signature. Any type that wants to be a `Shape` must have an `Area` method that returns a `float64`:

```
type Shape interface {
    Area() float64
}
```

Now we'll define two shapes, `Circle` and `Rectangle`, that satisfy the `Shape` interface by attaching an `Area` method to each one. Note that we don't have to explicitly declare that they satisfy the interface: if a type possesses all of its methods, it can *implicitly satisfy* an interface. This is particularly useful when you want to design interfaces that are satisfied by types that you don't own or control:

```
type Circle struct {
    Radius float64
}

func (c Circle) Area() float64 {
    return math.Pi * c.Radius * c.Radius
}

type Rectangle struct {
    Width, Height float64
}

func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}
```

Because both `Circle` and `Rectangle` implicitly satisfy the `Shape` interface, we can pass them to any function that expects a `Shape`:

```
func PrintArea(s Shape) {
    fmt.Printf("%T's area is %0.2f\n", s, s.Area())
}

func main() {
    r := Rectangle{Width:5, Height:10}
    PrintArea(r)                        // "main.Rectangle's area is 50.00"

    c := Circle{Radius:5}
    PrintArea(c)                        // "main.Circle's area is 78.54"
}
```

Type assertions

A *type assertion* can be applied to an interface value to "assert" its identity as a concrete type. The syntax takes the general form of `x.(T)`, where `x` is an expression of an interface, and `T` is the asserted type.

Referring to the `Shape` interface and `Circle` struct we used previously:

```
var s Shape
s = Circle{}              // s is an expression of Shape
c := s.(Circle)           // Assert that s is a Circle
fmt.Printf("%T\n", c)     // "main.Circle"
```

The empty interface

One curious construct is the *empty interface*: `interface{}`. The empty interface specifies no methods. It carries no information; it says nothing.[7]

A variable of type `interface{}` can hold values of any type, which can be very useful when your code needs to handle values of any type. The `fmt.Println` method is a good example of a function using this strategy.

There are downsides, however. Working with the empty interface requires certain assumptions to be made, which have to be checked at runtime and result in code that's more fragile and less efficient.

Composition with Type Embedding

Go doesn't allow subclassing or inheritance in the traditional object-oriented sense. Instead it allows types to be *embedded* within one another, extending the functionalities of the embedded types into the embedding type.

This is a particularly useful feature of Go that allows functionalities to be reused via *composition*—combining the features of existing types to create new types—instead of inheritance, removing the need for the kinds of elaborate type hierarchies that can saddle traditional object-oriented programming projects.

Interface embedding

A popular example of embedding interfaces comes to us by way of the `io` package. Specifically, the widely used `io.Reader` and `io.Writer` interfaces, which are defined as follows:

```
type Reader interface {
    Read(p []byte) (n int, err error)
}

type Writer interface {
    Write(p []byte) (n int, err error)
}
```

7 Pike, Rob. "Go Proverbs." YouTube. 1 Dec. 2015. *https://oreil.ly/g8Rid*.

But what if you want an interface with the methods of both an `io.Reader` and `io.Writer`? Well, you *could* implement a third interface that copies the methods of both, but then you have to keep all of them in agreement. That doesn't just add unnecessary maintenance overhead: it's also a good way to accidentally introduce errors.

Rather than go the copy–paste route, Go allows you to embed the two existing interfaces into a third one that takes on the features of both. Syntactically, this is done by adding the embedded interfaces as anonymous fields, as demonstrated by the standard `io.ReadWriter` interface, shown here:

```
type ReadWriter interface {
    Reader
    Writer
}
```

The result of this composition is a new interface that has all of the methods of the interfaces embedded within it.

Only interfaces can be embedded within interfaces.

Struct embedding

Embedding isn't limited to interfaces: structs can also be embedded into other structs.

The struct equivalent to the `io.Reader` and `io.Writer` example in the previous section comes from the `bufio` package. Specifically, `bufio.Reader` (which implements `io.Reader`) and `bufio.Writer` (which implements `io.Writer`). Similarly, `bufio` also provides an implementation of `io.ReadWriter`, which is just a composition of the existing `bufio.Reader` and `bufio.Writer` types:

```
type ReadWriter struct {
    *Reader
    *Writer
}
```

As you can see, the syntax for embedding structs is identical to that of interfaces: adding the embedded types as unnamed fields. In the preceding case, the `bufio.ReadWriter` embeds `bufio.Reader` and `bufio.Writer` as pointer types.

Just like any pointers, embedded pointers to structs have a zero value of `nil`, and must be initialized to point to valid structs before they can be used.

Promotion

So, why would you use composition instead of just adding a struct field? The answer is that when a type is embedded, its exported properties and methods are *promoted* to the embedding type, allowing them to be directly invoked. For example, the `Read` method of a `bufio.Reader` is accessible directly from an instance of `bufio.ReadWriter`:

```
var rw *bufio.ReadWriter = GetReadWriter()
var bytes []byte = make([]byte, 1024)

n, err := rw.Read(bytes) {
    // Do something
}
```

You don't have to know or care that the `Read` method is actually attached to the embedded `*bufio.Reader`. It's important to know, though, that when a promoted method is invoked the method's receiver is still the embedded type, so the receiver of `rw.Read` is the `ReadWriter`'s `Reader` field, not the `ReadWriter`.

Directly accessing embedded fields

Occasionally, you'll need to refer to an embedded field directly. To do this, you use the type name of the field as a field name. In the following (somewhat contrived) example, the `UseReader` function requires a `*bufio.Reader`, but what you have is a `*bufio.ReadWriter` instance:

```
func UseReader(r *bufio.Reader) {
    fmt.Printf("We got a %T\n", r)      // "We got a *bufio.Reader"
}

func main() {
    var rw *bufio.ReadWriter = GetReadWriter()
    UseReader(rw.Reader)
}
```

As you can see, this snippet uses the type name of the field you want to access (`Reader`) as the field name (`rw.Reader`) to retrieve the `*bufio.Reader` from `rw`. This can be handy for initialization as well:

```
rw := &bufio.ReadWriter{Reader: &bufio.Reader{}, Writer: &bufio.Writer{}}
```

If we'd just created `rw` as `&bufio.ReadWriter{}`, its embedded fields would be `nil`, but the snippet produces a `*bufio.ReadWriter` with fully defined `*bufio.Reader`

and `*bufio.Writer` fields. While you wouldn't typically do this with a `&bufio.ReadWriter`, this approach could be used to provide a useful mock in a pinch.

The Good Stuff: Concurrency

The intricacies of concurrent programming are many, and are well beyond the scope of this work. However, you can say that reasoning about concurrency is hard, and that the way concurrency is generally done makes it harder. In most languages, the usual approach to processes orchestration is to create some shared bit of memory, which is then wrapped in locks to restrict access to one process at a time, often introducing maddeningly difficult-to-debug errors such as race conditions or deadlocks.

Go, on the other hand, favors another strategy: it provides two concurrency primitives—goroutines and channels—that can be used together to elegantly structure concurrent software, that don't depend quite so much on locking. It encourages developers to limit sharing memory, and to instead allow processes to interact with one other entirely by passing messages.

Goroutines

One of Go's most powerful features is the `go` keyword. Any function call prepended with the `go` keyword will run as usual, but the caller can proceed uninterrupted rather than wait for the function to return. Under the hood, the function is executed as a lightweight, concurrently executing process called a *goroutine*.

The syntax is strikingly simple: a function `foo`, which may be executed sequentially as `foo()`, may be executed as a concurrent goroutine simply by adding the `go` keyword: `go foo()`:

```
foo()       // Call foo() and wait for it to return
go foo()    // Spawn a new goroutine that calls foo() concurrently
```

Goroutines can also be used to invoke a function literal:

```
func Log(w io.Writer, message string) {
    go func() {
        fmt.Fprintln(w, message)
    }() // Don't forget the trailing parentheses!
}
```

Channels

In Go, *channels* are typed primitives that allow communication between two goroutines. They act as pipes into which a value can be sent and then received by a goroutine on the other end.

Channels may be created using the `make` function. Each channel can transmit values of a specific type, called its *element type*. Channel types are written using the `chan` keyword followed by their element type. The following example declares and allocates an `int` channel:

```
var ch chan int = make(chan int)
```

The two primary operations supported by channels are *send* and *receive*, both of which use the `<-` operator, where the arrow indicates the direction of the data flow as demonstrated in the following:

```
ch <- val     // Sending on a channel
val = <-ch    // Receiving on a channel and assigning it to val
<-ch          // Receiving on a channel and discarding the result
```

Channel blocking

By default, a channel is *unbuffered*. Unbuffered channels have a very useful property: sends on them block until another goroutine receives on the channel, and receives block until another goroutine sends on the channel. This behavior can be exploited to synchronize two goroutines, as demonstrated in the following:

```
func main() {
    ch := make(chan string)    // Allocate a string channel

    go func() {
       message := <-ch         // Blocking receive; assigns to message
       fmt.Println(message)    // "ping"
       ch <- "pong"            // Blocking send
    }()

    ch <- "ping"               // Send "ping"
    fmt.Println(<-ch)          // "pong"
}
```

Although `main` and the anonymous `goroutine` run concurrently and could in theory run in any order, the blocking behavior of unbuffered channels guarantees that the output will always be "ping" followed by "pong."

Channel buffering

Go channels may be *buffered*, in which case they contain an internal value queue with a fixed *capacity* that's specified when the buffer is initialized. Sends to a buffered channel only block when the buffer is full; receives from a channel only block when the buffer is empty. Any other time, send and receive operations write to or read from the buffer, respectively, and exit immediately.

A buffered channel can be created by providing a second argument to the `make` function to indicate its capacity:

```
ch := make(chan string, 2)    // Buffered channel with capacity 2

ch <- "foo"                   // Two non-blocking sends
ch <- "bar"

fmt.Println(<-ch)             // Two non-blocking receives
fmt.Println(<-ch)

fmt.Println(<-ch)             // The third receive will block
```

Closing channels

The third available channel operation is *close*, which sets a flag to indicate that no more values will be sent on it. The built-in `close` function can be used to close a channel: `close(ch)`.

The channel close operation is just a flag to tell the receiver not to expect any more values. You don't *have to* explicitly close channels.

Trying to send on a closed channel will cause a `panic`. Receiving from a closed channel will retrieve any values sent on the channel prior to its closure; any subsequent receive operations will immediately yield the zero value of the channel's element type. Receivers may also test whether a channel has been closed (and its buffer is empty) by assigning a second `bool` parameter to the receive expression:

```
ch := make(chan string, 10)

ch <- "foo"

close(ch)                         // One value left in the buffer

msg, ok := <-ch
fmt.Printf("%q, %v\n", msg, ok)    // "foo", true

msg, ok = <-ch
fmt.Printf("%q, %v\n", msg, ok)    // "", false
```

While either party may close a channel, in practice only the sender should do so. Inadvertently sending on a closed channel will cause a panic.

Looping over channels

The `range` keyword may be used to loop over channels that are open or contain buffered values. The loop will block until a value is available to be read or until the channel is closed. You can see how this works in the following:

```
ch := make(chan string, 3)

ch <- "foo"                 // Send three (buffered) values to the channel
ch <- "bar"
ch <- "baz"

close(ch)                   // Close the channel

for s := range ch {         // Range will continue to the "closed" flag
    fmt.Println(s)
}
```

In this example, the buffered channel `ch` is created, and three values are sent before being closed. Because the three values were sent to the channel before it was closed, looping over this channel will output all three lines before terminating.

Had the channel not been closed, the loop would stop and wait for the next value to be sent along the channel, potentially indefinitely.

Select

Go's `select` statements are a little like `switch` statements that provide a convenient mechanism for multiplexing communications with multiple channels. The syntax for `select` is very similar to `switch`, with some number of `case` statements that specify code to be executed upon a successful send or receive operation:

```
select {
case <-ch1:                         // Discard received value
    fmt.Println("Got something")

case x := <-ch2:                    // Assign received value to x
    fmt.Println(x)

case ch3 <- y:                      // Send y to channel
    fmt.Println(y)

default:
    fmt.Println("None of the above")
}
```

In the preceding snippet, there are three primary cases specified with three different conditions. If the channel `ch1` is ready to be read, then its value will be read (and discarded) and the text "Got something" will be printed. If `ch2` is ready to be read, then its value will be read and assigned to the variable `x` before printing the value of `x`.

Finally, if `ch3` is ready to be sent to, then the value `y` is sent to it before printing the value of `y`.

Finally, if no cases are ready, the `default` statements will be executed. If there's no `default`, then the `select` will block until one of its cases is ready, at which point it performs the associated communication and executes the associated statements. If multiple cases are ready, `select` will execute one at random.

Gotcha!

When using `select`, keep in mind that a closed channel never blocks and is always readable.

Implementing channel timeouts

The ability to use `select` to multiplex on channels can be very powerful, and can make otherwise very difficult or tedious tasks trivial. Take, for example, the implementation of a timeout on an arbitrary channel. In some languages this might require some awkward thread work, but a `select` with a call to `time.After`, which returns a channel that sends a message after a specified duration, makes short work of it:

```
var ch chan int

select {
case m := <-ch:                     // Read from ch; blocks forever
    fmt.Println(m)

case <-time.After(10 * time.Second):   // time.After returns a channel
    fmt.Println("Timed out")
}
```

Since there's no `default` statement, this `select` will block until one if its case conditions becomes true. If `ch` doesn't become available to read before the channel returned by `time.After` emits a message, then the second case will activate and the statement will time out.

Summary

What I covered in this chapter could easily have consumed an entire book, if I'd been able to drill down into the level of detail the subject really deserves. But space and time are limited (and that book's already been written[8]) so I have to remain content to have only this one chapter as a broad and shallow survey of the Go language (at least until the second edition comes out).

But learning Go's syntax and grammar will only get you so far. In Chapter 4 I'll be presenting a variety of Go programming patterns that I see come up pretty regularly in the "cloud native" context. So, if you thought this chapter was interesting, you're going to love the next one.

8 One last time, if you haven't read it yet, go read *The Go Programming Language* by Donovan and Kernighan (Addison-Wesley Professional).

CHAPTER 4
Cloud Native Patterns

> Progress is possible only if we train ourselves to think about programs without thinking of them as pieces of executable code.[1]
>
> —Edsger W. Dijkstra, *August 1979*

In 1991, while still at Sun Microsystems, L Peter Deutsch[2] formulated the *Fallacies of Distributed Computing*, which lists some of the false assumptions that programmers new (and not so new) to distributed applications often make:

- *The network is reliable*: switches fail, routers get misconfigured
- *Latency is zero*: it takes time to move data across a network
- *Bandwidth is infinite*: a network can only handle so much data at a time
- *The network is secure*: don't share secrets in plain text; encrypt everything
- *Topology doesn't change*: servers and services come and go
- *There is one administrator*: multiple admins lead to heterogeneous solutions
- *Transport cost is zero*: moving data around costs time and money
- *The network is homogeneous*: every network is (sometimes very) different

If I might be so audacious, I'd like to add a ninth one as well:

- *Services are reliable*: services that you depend on can fail at any time

1 Spoken August 1979. Attested to by Vicki Almstrum, Tony Hoare, Niklaus Wirth, Wim Feijen, and Rajeev Joshi. In Pursuit of Simplicity: A Symposium Honoring Professor Edsger Wybe Dijkstra, 12–13 May 2000.

2 L (yes, his legal name is L) is a brilliant and fascinating human being. Look him up some time.

In this chapter, I'll present a selection of idiomatic patterns—tested, proven development paradigms—designed to address one or more of the conditions described in Deutsch's Fallacies, and demonstrate how to implement them in Go. None of the patterns discussed in this book are original to this book—some have been around for as long as distributed applications have existed—but most haven't been previously published together in a single work. Many of them are unique to Go or have novel implementations in Go relative to other languages.

Unfortunately, this book won't cover infrastructure-level patterns like the Bulkhead or Gatekeeper patterns. Largely, this is because our focus is on application-layer development in Go, and those patterns, while indispensable, function at an entirely different abstraction level. If you're interested in learning more, I recommend *Cloud Native Infrastructure* by Justin Garrison and Kris Nova (O'Reilly) and *Designing Distributed Systems* by Brendan Burns (O'Reilly).

The Context Package

Most of the code examples in this chapter make use of the `context` package, which was introduced in Go 1.7 to provide an idiomatic means of carrying deadlines, cancellation signals, and request-scoped values between processes. It contains a single interface, `context.Context`, whose methods are listed in the following:

```
type Context interface {
    // Done returns a channel that's closed when this Context is cancelled.
    Done() <-chan struct{}

    // Err indicates why this context was cancelled after the Done channel is
    // closed. If Done is not yet closed, Err returns nil.
    Err() error

    // Deadline returns the time when this Context should be cancelled; it
    // returns ok==false if no deadline is set.
    Deadline() (deadline time.Time, ok bool)

    // Value returns the value associated with this context for key, or nil
    // if no value is associated with key. Use with care.
    Value(key interface{}) interface{}
}
```

Three of these methods can be used to learn something about a `Context` value's cancellation status or behavior. The fourth, `Value`, can be used to retrieve a value associated with an arbitrary key. `Context`'s `Value` method is the focus of some controversy in the Go world, and will be discussed more in "Defining Request-Scoped Values" on page 75.

What Context Can Do for You

A `context.Context` value is used by passing it directly to a service request, which may in turn pass it to one or more subrequests. What makes this useful is that when the `Context` is cancelled, all functions holding it (or a derived `Context`; more on this in Figures 4-1, 4-2, and 4-3) will receive the signal, allowing them to coordinate their cancellation and reduce the amount of wasted effort.

Take, for example, a request from a user to a service, which in turn makes a request to a database. In an ideal scenario, the user, application, and database requests can be diagrammed as in Figure 4-1.

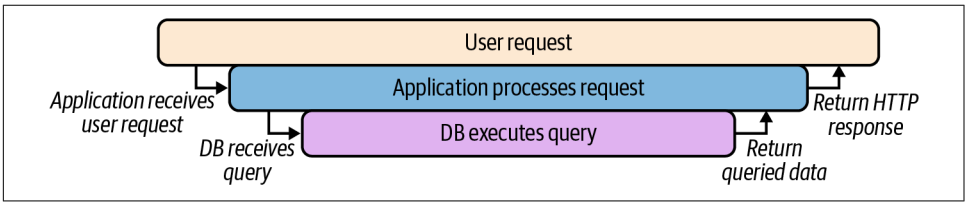

Figure 4-1. A successful request from a user, to a service, to a database

But what if the user terminates their request before it's fully completed? In most cases, oblivious to the overall context of the request, the processes will continue to live on anyway (Figure 4-2), consuming resources in order to provide a result that'll never be used.

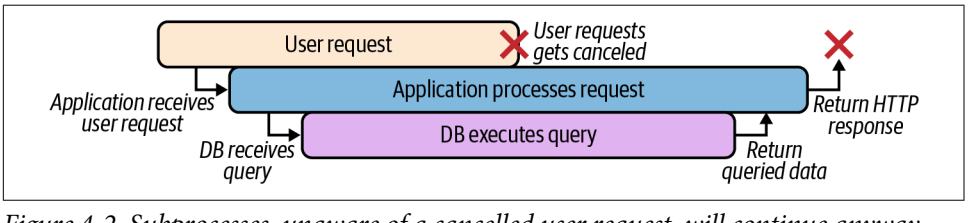

Figure 4-2. Subprocesses, unaware of a cancelled user request, will continue anyway

However, by sharing a `Context` to each subsequent request, all long-running processes can be sent a simultaneous "done" signal, allowing the cancellation signal to be coordinated among each of the processes (Figure 4-3).

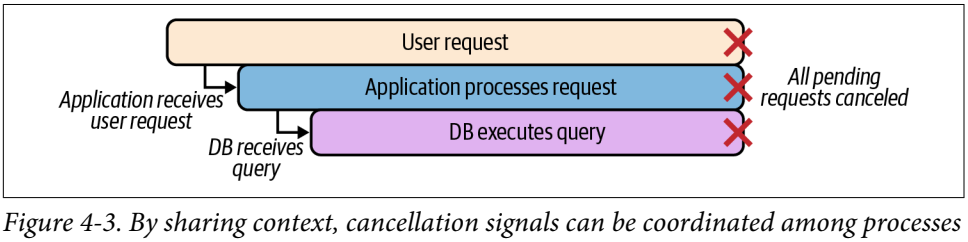

Figure 4-3. By sharing context, cancellation signals can be coordinated among processes

Importantly, `Context` values are also thread safe, i.e., they can be safely used by multiple concurrently executing goroutines without fear of unexpected behaviors.

Creating Context

A brand-new `context.Context` can be obtained using one of two functions:

`func Background() Context`
: Returns an empty `Context` that's never cancelled, has no values, and has no deadline. It is typically used by the main function, initialization, and tests, and as the top-level `Context` for incoming requests.

`func TODO() Context`
: Also provides an empty `Context`, but it's intended to be used as a placeholder when it's unclear which `Context` to use or when a parent `Context` is not yet available.

Defining Context Deadlines and Timeouts

The `context` package also includes a number of methods for creating *derived* `Context` values that allow you to direct cancellation behavior, either by applying a timeout or by a function hook that can explicitly trigger a cancellation.

`func WithDeadline(Context, time.Time) (Context, CancelFunc)`
: Accepts a specific time at which the `Context` will be cancelled and the `Done` channel will be closed.

`func WithTimeout(Context, time.Duration) (Context, CancelFunc)`
: Accepts a duration after which the `Context` will be cancelled and the `Done` channel will be closed.

`func WithCancel(Context) (Context, CancelFunc)`
: Unlike the previous functions, `WithCancel` accepts nothing, and only returns a function that can be called to explicitly cancel the `Context`.

All three of these functions return a derived `Context` that includes any requested decoration, and a `context.CancelFunc`, a zero-parameter function that can be called to explicitly cancel the `Context` and all of its derived values.

When a `Context` is cancelled, all `Contexts` that are *derived from it* are also cancelled. `Contexts` that *it was derived from* are not.

Defining Request-Scoped Values

Finally, the `context` package includes a function that can be used to define an arbitrary *request-scoped* key-value pair that can be accessed from the returned `Context`—and all `Context` values derived from it—via the `Value` method.

`func WithValue(parent Context, key, val interface{}) Context`

`WithValue` returns a derivation of `parent` in which `key` is associated with the value `val`.

On Context Values

The `context.WithValue` and `context.Value` functions provide convenient mechanisms for setting and getting arbitrary key-value pairs that can be used by consuming processes and APIs. However, it has been argued that this functionality is orthogonal to `Context`'s function of orchestrating the cancellation of long-lived requests, obscures your program's flow, and can easily break compile-time coupling. For a more in-depth discussion, please see Dave Cheney's blog post *Context Is for Cancelation*.

This functionality isn't used in any of the examples in this chapter (or this book). If you choose to make use of it, please take care to ensure that all of your values are scoped only to the request, don't alter the functioning of any processes, and don't break your processes if they happen to be absent.

Using a Context

When a service request is initiated, either by an incoming request or triggered by the `main` function, the top-level process will use the `Background` function to create a new `Context` value, possibly decorating it with one or more of the `context.With*` functions, before passing it along to any subrequests. Those subrequests then need only watch the `Done` channel for cancellation signals.

For example, take a look at the following `Stream` function:

```
func Stream(ctx context.Context, out chan<- Value) error {
    // Create a derived Context with a 10s timeout; dctx
    // will be cancelled upon timeout, but ctx will not.
    // cancel is a function that will explicitly cancel dctx.
    dctx, cancel := context.WithTimeout(ctx, time.Second * 10)

    // Release resources if SlowOperation completes before timeout
    defer cancel()

    res, err := SlowOperation(dctx)
    if err != nil {                    // True if dctx times out
```

```
            return err
        }

        for {
            select {
            case out <- res:                // Read from res; send to out

            case <-ctx.Done():              // Triggered if ctx is cancelled
                return ctx.Err()
            }
        }
    }
```

`Stream` receives a `ctx Context` as an input parameter, which it sends to `WithTimeout` to create `dctx`, a derived `Context` with a 10-second timeout. Because of this decoration, the `SlowOperation(dctx)` call could possibly time out after ten seconds and return an error. Functions using the original `ctx`, however, will not have this timeout decoration, and will not time out.

Further down, the original `ctx` value is used in a `for` loop around a `select` statement to retrieve values from the `res` channel provided by the `SlowOperation` function. Note the `case <-ctx.Done()` statement, which is executed when the `ctx.Done` channel closes to return an appropriate error value.

Layout of this Chapter

The general presentation of each pattern in this chapter is loosely based on the one used in the famous "Gang of Four" *Design Patterns* book,[3] but simpler and less formal. Each pattern opens with a very brief description of its purpose and the reasons for using it, and is followed by the following sections:

Applicability
: Context and descriptions of where this pattern may be applied.

Participants
: A listing of the components of the pattern and their roles.

Implementation
: A discussion of the solution and its implementation.

Sample code
: A demonstration of how the code may be implemented in Go.

3 Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st edition. Addison-Wesley Professional, 1994).

Stability Patterns

The stability patterns presented here address one or more of the assumptions called out by the Fallacies of Distributed Computing. They're generally intended to be applied by distributed applications to improve their own stability and the stability of the larger system they're a part of.

Circuit Breaker

Circuit Breaker automatically degrades service functions in response to a likely fault, preventing larger or cascading failures by eliminating recurring errors and providing reasonable error responses.

Applicability

If the *Fallacies of Distributed Computing* were to be distilled to one point, it would be that errors and failures are an undeniable fact of life for distributed, cloud native systems. Services become misconfigured, databases crash, networks partition. We can't prevent it; we can only accept and account for it.

Failing to do so can have some rather unpleasant consequences. We've all seen them, and they aren't pretty. Some services might keep futilely trying to do their job and returning nonsense to their client; others might fail catastrophically and maybe even fall into a crash/restart death spiral. It doesn't matter, because in the end they're all wasting resources, obscuring the source of original failure, and making cascading failures even more likely.

On the other hand, a service that's designed with the assumption that its dependencies can fail at any time can respond reasonably when they do. The Circuit Breaker allows a service to detect such failures and to "open the circuit" by temporarily ceasing to execute requests, instead providing clients with an error message consistent with the service's communication contract.

For example, imagine a service that (ideally) receives a request from a client, executes a database query, and returns a response. What if the database fails? The service might continue futilely trying to query it anyway, flooding the logs with error messages and eventually timing out or returning useless errors. Such a service can use a Circuit Breaker to "open the circuit" when the database fails, preventing the service from making any more doomed database requests (at least for a while), and allowing it to respond to the client immediately with a meaningful notification.

Participants

This pattern includes the following participants:

Circuit
: The function that interacts with the service.

Breaker
: A closure with the same function signature as *Circuit*.

Implementation

Essentially, the Circuit Breaker is just a specialized Adapter pattern, with `Breaker` wrapping `Circuit` to add some additional error handling logic.

Like the electrical switch from which this pattern derives its name, `Breaker` has two possible states: *closed* and *open*. In the closed state everything is functioning normally. All requests received from the client by `Breaker` are forwarded unchanged to `Circuit`, and all responses from `Circuit` are forwarded back to the client. In the open state, `Breaker` doesn't forward requests to `Circuit`. Instead it "fails fast" by responding with an informative error message.

`Breaker` internally tracks the errors returned by `Circuit`; if the number of consecutive errors returned by `Circuit` returns exceeds a defined threshold, `Breaker` *trips* and its state switches to *open*.

Most implementations of Circuit Breaker include some logic to automatically close the circuit after some period of time. Keep in mind, though, that hammering an already malfunctioning service with lots of retries can cause its own problems, so it's standard to include some kind of *backoff*, logic that reduces the rate of retries over time. The subject of backoff is actually fairly nuanced, but it will be covered in detail in in "Play It Again: Retrying Requests" on page 275.

In a multinode service, this implementation may be extended to include some shared storage mechanism, such as a Memcached or Redis network cache, to track the circuit state.

Sample code

We begin by creating a `Circuit` type that specifies the signature of the function that's interacting with your database or other upstream service. In practice, this can take whatever form is appropriate for your functionality. It should include an `error` in its return list, however:

```
type Circuit func(context.Context) (string, error)
```

In this example, `Circuit` is a function that accepts a `Context` value, which was described in depth in "The Context Package" on page 72. Your implementation may vary.

The `Breaker` function accepts any function that conforms to the `Circuit` type definition, and an unsigned integer representing the number of consecutive failures allowed before the circuit automatically opens. In return it provides another function, which also conforms to the `Circuit` type definition:

```
func Breaker(circuit Circuit, failureThreshold uint) Circuit {
    var consecutiveFailures int = 0
    var lastAttempt = time.Now()
    var m sync.RWMutex

    return func(ctx context.Context) (string, error) {
        m.RLock()                       // Establish a "read lock"

        d := consecutiveFailures - int(failureThreshold)

        if d >= 0 {
            shouldRetryAt := lastAttempt.Add(time.Second * 2 << d)
            if !time.Now().After(shouldRetryAt) {
                m.RUnlock()
                return "", errors.New("service unreachable")
            }
        }

        m.RUnlock()                     // Release read lock

        response, err := circuit(ctx)   // Issue request proper

        m.Lock()                        // Lock around shared resources
        defer m.Unlock()

        lastAttempt = time.Now()        // Record time of attempt

        if err != nil {                 // Circuit returned an error,
            consecutiveFailures++       // so we count the failure
            return response, err        // and return
        }

        consecutiveFailures = 0         // Reset failures counter

        return response, nil
    }
}
```

The `Breaker` function constructs another function, also of type `Circuit`, which wraps `circuit` to provide the desired functionality. You may recognize this from "Anonymous Functions and Closures" on page 55 as a closure: a nested function with access

to the variables of its parent function. As you will see, all of the "stability" functions implemented for this chapter work this way.

The closure works by counting the number of consecutive errors returned by `circuit`. If that value meets the failure threshold, then it returns the error "service unreachable" without actually calling `circuit`. Any successful calls to `circuit` cause `consecutiveFailures` to reset to 0, and the cycle begins again.

The closure even includes an automatic reset mechanism that allows requests to call `circuit` again after several seconds, with an *exponential backoff* in which the durations of the delays between retries roughly doubles with each attempt. Though simple and quite common, this actually isn't the ideal backoff algorithm. We'll review exactly why in "Backoff Algorithms" on page 276.

Debounce

Debounce limits the frequency of a function invocation so that only the first or last in a cluster of calls is actually performed.

Applicability

Debounce is the second of our patterns to be labeled with an electrical circuit theme. Specifically, it's named after a phenomenon in which a switch's contacts "bounce" when they're opened or closed, causing the circuit to fluctuate a bit before settling down. It's usually no big deal, but this "contact bounce" can be a real problem in logic circuits where a series of on/off pulses can be interpreted as a data stream. The practice of eliminating contact bounce so that only one signal is transmitted by an opening or closing contact is called "debouncing."

In the world of services, we sometimes find ourselves performing a cluster of potentially slow or costly operations where only one would do. Using the Debounce pattern, a series of similar calls that are tightly clustered in time are restricted to only one call, typically the first or last in a batch.

This technique has been used in the JavaScript world for years to limit the number of operations that could slow the browser by taking only the first in a series of user events or to delay a call until a user is ready. You've probably seen an application of this technique in practice before. We're all familiar with the experience of using a search bar whose autocomplete pop-up doesn't display until after you pause typing, or spam-clicking a button only to see the clicks after the first ignored.

Those of us who specialize in backend services can learn a lot from our frontend brethren, who have been working for years to account for the reliability, latency, and bandwidth issues inherent to distributed systems. For example, this approach could be used to retrieve some slowly updating remote resource without bogging down, wasting both client and server time with wasteful requests.

This pattern is similar to "Throttle" on page 86, in that it limits how often a function can be called. But where Debounce restricts clusters of invocations, Throttle simply limits according to time period. For more on the difference between the Debounce and Throttle patterns, see "What's the Difference Between Throttle and Debounce?" on page 87.

Participants

This pattern includes the following participants:

Circuit
: The function to regulate.

Debounce
: A closure with the same function signature as *Circuit*.

Implementation

The Debounce implementation is actually very similar to the one for Circuit Breaker in that it wraps *Circuit* to provide the rate-limiting logic. That logic is actually quite straightforward: on each call of the outer function—regardless of its outcome—a time interval is set. Any subsequent call made before that time interval expires is ignored; any call made afterwards is passed along to the inner function. This implementation, in which the inner function is called once and subsequent calls are ignored, is called *function-first*, and is useful because it allows the initial response from the inner function to be cached and returned.

A *function-last* implementation will wait for a pause after a series of calls before calling the inner function. This variant is common in the JavaScript world when a programmer wants a certain amount of input before making a function call, such as when a search bar waits for a pause in typing before autocompleting. Function-last tends to be less common in backend services because it doesn't provide an immediate response, but it can be useful if your function doesn't need results right away.

Sample code

Just like in the Circuit Breaker implementation, we start by defining a function type with the signature of the function we want to limit. Also like Circuit Breaker, we call it `Circuit`; it's identical to the one declared in that example. Again, `Circuit` can take whatever form is appropriate for your functionality, but it should include an `error` in its returns:

```
type Circuit func(context.Context) (string, error)
```

The similarity with the Circuit Breaker implementation is quite intentional: their compatibility makes them "chainable," as demonstrated in the following:

```
func myFunction func(ctx context.Context) (string, error) { /* ... */ }

wrapped := Breaker(Debounce(myFunction))
response, err := wrapped(ctx)
```

The function-first implementation of Debounce—`DebounceFirst`—is very straightforward compared to function-last because it only needs to track the last time it was called and return a cached result if it's called again less than `d` duration after:

```
func DebounceFirst(circuit Circuit, d time.Duration) Circuit {
    var threshold time.Time
    var result string
    var err error
    var m sync.Mutex

    return func(ctx context.Context) (string, error) {
        m.Lock()

        defer func() {
            threshold = time.Now().Add(d)
            m.Unlock()
        }()

        if time.Now().Before(threshold) {
            return result, err
        }

        result, err = circuit(ctx)

        return result, err
    }
}
```

This implementation of `DebounceFirst` takes pains to ensure thread safety by wrapping the entire function in a mutex. While this will force overlapping calls at the start of a cluster to have to wait until the result is cached, it also guarantees that `circuit` is called exactly once, at the very beginning of a cluster. A `defer` ensures that the value of `threshold`, representing the time when a cluster ends (if there are no further calls), is reset with every call.

Our function-last implementation is a bit more awkward because it involves the use of a `time.Ticker` to determine whether enough time has passed since the function was last called, and to call `circuit` when it has. Alternatively, we could create a new `time.Ticker` with every call, but that can get quite expensive if it's called frequently:

```
type Circuit func(context.Context) (string, error)

func DebounceLast(circuit Circuit, d time.Duration) Circuit {
    var threshold time.Time = time.Now()
    var ticker *time.Ticker
    var result string
    var err error
    var once sync.Once
    var m sync.Mutex

    return func(ctx context.Context) (string, error) {
        m.Lock()
        defer m.Unlock()

        threshold = time.Now().Add(d)

        once.Do(func() {
            ticker = time.NewTicker(time.Millisecond * 100)

            go func() {
                defer func() {
                    m.Lock()
                    ticker.Stop()
                    once = sync.Once{}
                    m.Unlock()
                }()

                for {
                    select {
                    case <-ticker.C:
                        m.Lock()
                        if time.Now().After(threshold) {
                            result, err = circuit(ctx)
                            m.Unlock()
                            return
                        }
                        m.Unlock()
                    case <-ctx.Done():
                        m.Lock()
                        result, err = "", ctx.Err()
                        m.Unlock()
                        return
                    }
                }
            }()
        })

        return result, err
    }
}
```

Like `DebounceFirst`, `DebounceLast` uses a value called `threshold` to indicate the end of a cluster of calls (assuming there are no additional calls). The similarity largely ends there however.

You'll notice that almost the entire function is run inside of the `Do` method of a `sync.Once` value, which ensures that (as its name suggests) the contained function is run *exactly* once. Inside this block, a `time.Ticker` is used to check whether `threshold` has been passed and to call `circuit` if it has. Finally, the `time.Ticker` is stopped, the `sync.Once` is reset, and the cycle is primed to repeat.

Retry

Retry accounts for a possible transient fault in a distributed system by transparently retrying a failed operation.

Applicability

Transient errors are a fact of life when working with complex distributed systems. These can be caused by any number of (hopefully) temporary conditions, especially if the downstream service or network resource has protective strategies in place, such as throttling that temporarily rejects requests under high workload, or adaptive strategies like autoscaling that can add capacity when needed.

These faults typically resolve themselves after a bit of time, so repeating the request after a reasonable delay is likely (but not guaranteed) to be successful. Failing to account for transient faults can lead to a system that's unnecessarily brittle. On the other hand, implementing an automatic retry strategy can considerably improve the stability of the service that can benefit both it and its upstream consumers.

Participants

This pattern includes the following participants:

Effector
: The function that interacts with the service.

Retry
: A function that accepts *Effector* and returns a closure with the same function signature as *Effector*.

Implementation

This pattern works similarly to Circuit Breaker or Debounce in that there is a type, *Effector*, that defines a function signature. This signature can take whatever form is appropriate for your implementation, but when the function executing the

potentially-failing operation is implemented, it must match the signature defined by *Effector*.

The *Retry* function accepts the user-defined *Effector* function and returns an *Effector* function that wraps the user-defined function to provide the retry logic. Along with the user-defined function, *Retry* also accepts an integer describing the maximum number of retry attempts that it will make, and a `time.Duration` that describes how long it'll wait between each retry attempt. If the `retries` parameter is 0, then the retry logic will effectively become a no-op.

Although not included here, retry logic will typically include some kind of a backoff algorithm.

Sample code

The signature for function argument of the `Retry` function is `Effector`. It looks exactly like the function types for the previous patterns:

```
type Effector func(context.Context) (string, error)
```

The `Retry` function itself is relatively straightforward, at least when compared to the functions we've seen so far:

```
func Retry(effector Effector, retries int, delay time.Duration) Effector {
    return func(ctx context.Context) (string, error) {
        for r := 0; ; r++ {
            response, err := effector(ctx)
            if err == nil || r >= retries {
                return response, err
            }

            log.Printf("Attempt %d failed; retrying in %v", r + 1, delay)

            select {
            case <-time.After(delay):
            case <-ctx.Done():
                return "", ctx.Err()
            }
        }
    }
}
```

You may have already noticed what it is that keeps the `Retry` function so slender: although it returns a function, that function doesn't have any external state. This means we don't need any elaborate mechanisms to support concurrency.

To use `Retry`, we can implement the function that executes the potentially-failing operation and whose signature matches the `Effector` type; this role is played by `EmulateTransientError` in the following example:

```
var count int

func EmulateTransientError(ctx context.Context) (string, error) {
    count++

    if count <= 3 {
        return "intentional fail", errors.New("error")
    } else {
        return "success", nil
    }
}

func main() {
    r := Retry(EmulateTransientError, 5, 2*time.Second)

    res, err := r(context.Background())

    fmt.Println(res, err)
}
```

In the `main` function, the `EmulateTransientError` function is passed to `Retry`, providing the function variable `r`. When `r` is called, `EmulateTransientError` is called, and called again after a delay if it returns an error, according to the retry logic shown previously. Finally, after the fourth attempt, `EmulateTransientError` returns a `nil` error and exits.

Throttle

Throttle limits the frequency of a function call to some maximum number of invocations per unit of time.

Applicability

The Throttle pattern is named after a device used to manage the flow of a fluid, such as the amount of fuel going into a car engine. Like its namesake mechanism, Throttle restricts the number of times that a function can be called during over a period of time. For example:

- A user may only be allowed 10 service requests per second.
- A client may restrict itself to call a particular function once every 500 milliseconds.
- An account may only be allowed three failed login attempts in a 24-hour period.

Perhaps the most common reason to apply a Throttle is to account for sharp activity spikes that could saturate the system with a possibly unreasonable number of requests that may be expensive to satisfy, or lead to service degradation and eventually failure. While it may be possible for a system to scale up to add sufficient capacity to meet user demand, this takes time, and the system may not be able to react quickly enough.

What's the Difference Between Throttle and Debounce?

Conceptually, Debounce and Throttle seem fairly similar. After all, they're both about reducing the number of calls per unit of time. However, as illustrated in Figure 4-4, the precise timing of each differs quite a bit:

- *Throttle* works like the throttle in a car, limiting the amount of fuel going into the engine by capping the flow of fuel to some maximum rate. This is illustrated in Figure 4-4: no matter how many times the input function is called, Throttle only allows a fixed number of calls to proceed per unit of time.
- *Debounce* focuses on clusters of activity, making sure that a function is called only once during a cluster of requests, either at the start or the end of the cluster. A function-first debounce implementation is illustrated in Figure 4-4: for each of the two clusters of calls to the input function, Debounce only allows one call to proceed at the beginning of each cluster.

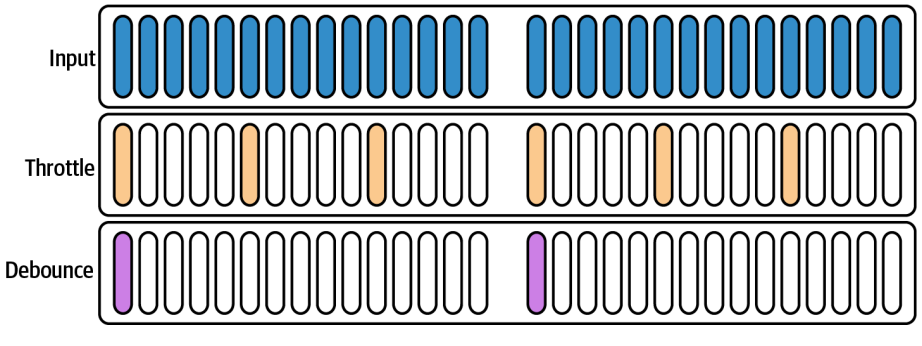

Figure 4-4. Throttle limits the event rate; debounce allows only one event in a cluster

Participants

This pattern includes the following participants:

Effector
: The function to regulate.

Throttle
: A function that accepts *Effector* and returns a closure with the same function signature as *Effector*.

Implementation

The Throttle pattern is similar to many of the other patterns described in this chapter: it's implemented as a function that accepts an effector function, and returns a `Throttle` closure with the same signature that provides the rate-limiting logic.

The most common algorithm for implementing rate-limiting behavior is the *token bucket*, which uses the analogy of a bucket that can hold some maximum number of tokens. When a function is called, a token is taken from the bucket, which then refills at some fixed rate.

The way that a `Throttle` treats requests when there are insufficient tokens in the bucket to pay for it can vary depending according to the needs of the developer. Some common strategies are:

Return an error
: This is the most basic strategy and is common when you're only trying to restrict unreasonable or potentially abusive numbers of client requests. A RESTful service adopting this strategy might respond with a status `429 (Too Many Requests)`.

Replay the response of the last successful function call
: This strategy can be useful when a service or expensive function call is likely to provide an identical result if called too soon. It's commonly used in the JavaScript world.

Enqueue the request for execution when sufficient tokens are available
: This approach can be useful when you want to eventually handle all requests, but it's also more complex and may require care to be taken to ensure that memory isn't exhausted.

Sample code

The following example implements a very basic "token bucket" algorithm that uses the "error" strategy:

```
type Effector func(context.Context) (string, error)

func Throttle(e Effector, max uint, refill uint, d time.Duration) Effector {
    var tokens = max
    var once sync.Once

    return func(ctx context.Context) (string, error) {
        if ctx.Err() != nil {
            return "", ctx.Err()
        }

        once.Do(func() {
            ticker := time.NewTicker(d)

            go func() {
                defer ticker.Stop()

                for {
                    select {
                    case <-ctx.Done():
                        return

                    case <-ticker.C:
                        t := tokens + refill
                        if t > max {
                            t = max
                        }
                        tokens = t
                    }
                }
            }()
        })

        if tokens <= 0 {
            return "", fmt.Errorf("too many calls")
        }

        tokens--

        return e(ctx)
    }
}
```

This `Throttle` implementation is similar to our other examples in that it wraps an effector function `e` with a closure that contains the rate-limiting logic. The bucket is initially allocated `max` tokens; each time the closure is triggered it checks whether it has any remaining tokens. If tokens are available, it decrements the token count by one and triggers the effector function. If not, an error is returned. Tokens are added at a rate of `refill` tokens every duration `d`.

Timeout

Timeout allows a process to stop waiting for an answer once it's clear that an answer may not be coming.

Applicability

The first of the Fallacies of Distributed Computing is that "the network is reliable," and it's first for a reason. Switches fail, routers and firewalls get misconfigured; packets get blackholed. Even if your network is working perfectly, not every service is thoughtful enough to guarantee a meaningful and timely response—or any response at all—if and when it malfunctions.

Timeout represents a common solution to this dilemma, and is so beautifully simple that it barely even qualifies as a pattern at all: given a service request or function call that's running for a longer-than-expected time, the caller simply…stops waiting.

However, don't mistake "simple" or "common" for "useless." On the contrary, the ubiquity of the timeout strategy is a testament to its usefulness. The judicious use of timeouts can provide a degree of fault isolation, preventing cascading failures and reducing the chance that a problem in a downstream resource becomes *your* problem.

Participants

This pattern includes the following participants:

Client
: The client who wants to execute *SlowFunction*.

SlowFunction
: The long-running function that implements the functionality desired by *Client*.

Timeout
: A wrapper function around *SlowFunction* that implements the timeout logic.

Implementation

There are several ways to implement a timeout in Go, but the idiomatic way is to use the functionality provided by the `context` package. See "The Context Package" on page 72 for more information.

In an ideal world, any possibly long-running function will accept a `context.Context` parameter directly. If so, your work is fairly straightforward: you need only pass it a `Context` value decorated with the `context.WithTimeout` function:

```
ctx := context.Background()
ctxt, cancel := context.WithTimeout(ctx, 10 * time.Second)
defer cancel()

result, err := SomeFunction(ctxt)
```

However, this isn't always the case, and with third party libraries you don't always have the option of refactoring to accept a `Context` value. In these cases, the best course of action may be to wrap the function call in such a way that it *does* respect your `Context`.

For example, imagine you have a potentially long-running function that not only doesn't accept a `Context` value, but comes from a package you don't control. If *Client* were to call *SlowFunction* directly it would be forced to wait until the function completes, if indeed it ever does. Now what?

Instead of calling *SlowFunction* directly, you can call it in a goroutine. In this way, you can capture the results it returns, if it returns them in an acceptable period of time. However, this also allows you to move on if it doesn't.

To do this, we can leverage a few tools that we've seen before: `context.Context` for timeouts, channels for communicating results, and `select` to catch whichever one acts first.

Sample code

The following example imagines the existence of the fictional function, `Slow`, whose execution may or may not complete in some reasonable amount of time, and whose signature conforms with the following type definition:

```
type SlowFunction func(string) (string, error)
```

Rather than calling `Slow` directly, we instead provide a `Timeout` function, which wraps a provided `SlowFunction` in a closure and returns a `WithContext` function, which adds a `context.Context` to the `SlowFunction`'s parameter list:

```
type WithContext func(context.Context, string) (string, error)

func Timeout(f SlowFunction) WithContext {
    return func(ctx context.Context, arg string) (string, error) {
        chres := make(chan string)
        cherr := make(chan error)

        go func() {
            res, err := f(arg)
            chres <- res
            cherr <- err
        }()

        select {
```

```
            case res := <-chres:
                return res, <-cherr
            case <-ctx.Done():
                return "", ctx.Err()
            }
        }
    }
```

Within the function that `Timeout` constructs, `Slow` is run in a goroutine, with its return values being sent into channels constructed for that purpose, if and when it ever completes.

The following goroutine statement is a `select` block on two channels: the first of the `Slow` function response channels, and the `Context` value's `Done` channel. If the former completes first, the closure will return the `Slow` function's return values; otherwise it returns the error provided by the `Context`.

Using the `Timeout` function isn't much more complicated than consuming `Slow` directly, except that instead of one function call, we have two: the call to `Timeout` to retrieve the closure, and the call to the closure itself:

```
func main() {
    ctx := context.Background()
    ctxt, cancel := context.WithTimeout(ctx, 1*time.Second)
    defer cancel()

    timeout := Timeout(Slow)
    res, err := timeout(ctxt, "some input")

    fmt.Println(res, err)
}
```

Finally, although it's usually preferred to implement service timeouts using `context.Context`, channel timeouts *can* also be implemented using the channel provided by the `time.After` function. See "Implementing channel timeouts" on page 68 for an example of how this is done.

Concurrency Patterns

A cloud native service will often be called upon to efficiently juggle multiple processes and handle high (and highly variable) levels of load, ideally without having to suffer the trouble and expense of scaling up. As such, it needs to be highly concurrent and able to manage multiple simultaneous requests from multiple clients. While Go is known for its concurrency support, bottlenecks can and do happen. Some of the patterns that have been developed to prevent them are presented here.

Fan-In

Fan-in multiplexes multiple input channels onto one output channel.

Applicability

Services that have some number of workers that all generate output may find it useful to combine all of the workers' outputs to be processed as a single unified stream. For these scenarios we use the fan-in pattern, which can read from multiple input channels by multiplexing them onto a single destination channel.

Participants

This pattern includes the following participants:

Sources
: A set of one or more input channels with the same type. Accepted by *Funnel*.

Destination
: An output channel of the same type as *Sources*. Created and provided by *Funnel*.

Funnel
: Accepts *Sources* and immediately returns *Destination*. Any input from any *Sources* will be output by *Destination*.

Implementation

Funnel is implemented as a function that receives zero to N input channels (*Sources*). For each input channel in *Sources*, the *Funnel* function starts a separate goroutine to read values from its assigned channel and forward them to a single output channel shared by all of the goroutines (*Destination*).

Sample code

The `Funnel` function is a variadic function that receives `sources`: zero to N channels of some type (`int` in the following example):

```
func Funnel(sources ...<-chan int) <-chan int {
    dest := make(chan int)                  // The shared output channel

    var wg sync.WaitGroup                   // Used to automatically close dest
                                            // when all sources are closed

    wg.Add(len(sources))                    // Set size of the WaitGroup

    for _, ch := range sources {            // Start a goroutine for each source
        go func(c <-chan int) {
            defer wg.Done()                 // Notify WaitGroup when c closes
```

```
            for n := range c {
                dest <- n
            }
        }(ch)
    }

    go func() {                             // Start a goroutine to close dest
        wg.Wait()                           // after all sources close
        close(dest)
    }()

    return dest
}
```

For each channel in the list of `sources`, `Funnel` starts a dedicated goroutine that reads values from its assigned channel and forwards them to `dest`, a single-output channel shared by all of the goroutines.

Note the use of a `sync.WaitGroup` to ensure that the destination channel is closed appropriately. Initially, a `WaitGroup` is created and set to the total number of source channels. If a channel is closed, its associated goroutine exits, calling `wg.Done`. When all of the channels are closed, the WaitGroup's counter reaches zero, the lock imposed by `wg.Wait` is released, and the `dest` channel is closed.

Using `Funnel` is reasonably straightforward: given N source channels (or a slice of N channels), pass the channels to `Funnel`. The returned destination channel may be read in the usual way, and will close when all source channels close:

```
func main() {
    sources := make([]<-chan int, 0)        // Create an empty channel slice

    for i := 0; i < 3; i++ {
        ch := make(chan int)
        sources = append(sources, ch)       // Create a channel; add to sources

        go func() {                         // Run a toy goroutine for each
            defer close(ch)                 // Close ch when the routine ends

            for i := 1; i <= 5; i++ {
                ch <- i
                time.Sleep(time.Second)
            }
        }()
    }

    dest := Funnel(sources...)
    for d := range dest {
        fmt.Println(d)
    }
}
```

This example creates a slice of three `int` channels, into which the values from 1 to 5 are sent before being closed. In a separate goroutine, the outputs of the single `dest` channel are printed. Running this will result in the appropriate 15 lines being printed before `dest` closes and the function ends.

Fan-Out

Fan-out evenly distributes messages from an input channel to multiple output channels.

Applicability

Fan-out receives messages from an input channel, distributing them evenly among output channels, and is a useful pattern for parallelizing CPU and I/O utilization.

For example, imagine that you have an input source, such as a `Reader` on an input stream, or a listener on a message broker, that provides the inputs for some resource-intensive unit of work. Rather than coupling the input and computation processes, which would confine the effort to a single serial process, you might prefer to parallelize the workload by distributing it among some number of concurrent worker processes.

Participants

This pattern includes the following participants:

Source
: An input channel. Accepted by *Split*.

Destinations
: An output channel of the same type as *Source*. Created and provided by *Split*.

Split
: A function that accepts *Source* and immediately returns *Destinations*. Any input from *Source* will be output to a *Destination*.

Implementation

Fan-out may be relatively conceptually straightforward, but the devil is in the details.

Typically, fan-out is implemented as a *Split* function, which accepts a single *Source* channel and integer representing the desired number of *Destination* channels. The *Split* function creates the *Destination* channels and executes some background process that retrieves values from *Source* channel and forwards them to one of the *Destinations*.

The implementation of the forwarding logic can be done in one of two ways:

- Using a single goroutine that reads values from *Source* and forwards them to the *Destinations* in a round-robin fashion. This has the virtue of requiring only one master goroutine, but if the next channel isn't ready to read yet, it'll slow the entire process.
- Using separate goroutines for each *Destination* that compete to read the next value from *Source* and forward it to their respective *Destination*. This requires slightly more resources, but is less likely to get bogged down by a single slow-running worker.

The next example uses the latter approach.

Sample code

In this example, the `Split` function accepts a single receive-only channel, `source`, and an integer describing the number of channels to split the input into, `n`. It returns a slice of `n` send-only channels with the same type as `source`.

Internally, `Split` creates the destination channels. For each channel created, it executes a goroutine that retrieves values from `source` in a `for` loop and forwards them to their assigned output channel. Effectively, each goroutine competes for reads from `source`; if several are trying to read, the "winner" will be randomly determined. If `source` is closed, all goroutines terminate and all of the destination channels are closed:

```
func Split(source <-chan int, n int) []<-chan int {
    dests := make([]<-chan int, 0)          // Create the dests slice

    for i := 0; i < n; i++ {                // Create n destination channels
        ch := make(chan int)
        dests = append(dests, ch)

        go func() {                         // Each channel gets a dedicated
            defer close(ch)                 // goroutine that competes for reads

            for val := range source {
                ch <- val
            }
        }()
    }

    return dests
}
```

Given a channel of some specific type, the `Split` function will return a number of destination channels. Typically, each will be passed to a separate goroutine, as demonstrated in the following example:

```
func main() {
    source := make(chan int)               // The input channel
    dests := Split(source, 5)              // Retrieve 5 output channels

    go func() {                            // Send the number 1..10 to source
        for i := 1; i <= 10; i++ {         // and close it when we're done
            source <- i
        }

        close(source)
    }()

    var wg sync.WaitGroup                  // Use WaitGroup to wait until
    wg.Add(len(dests))                     // the output channels all close

    for i, ch := range dests {
        go func(i int, d <-chan int) {
            defer wg.Done()

            for val := range d {
                fmt.Printf("#%d got %d\n", i, val)
            }
        }(i, ch)
    }

    wg.Wait()
}
```

This example creates an input channel, `source`, which it passes to `Split` to receive its output channels. Concurrently it passes the values 1 to 10 into `source` in a goroutine, while receiving values from `dests` in five others. When the inputs are complete, the `source` channel is closed, which triggers closures in the output channels, which ends the read loops, which causes `wg.Done` to be called by each of the read goroutines, which releases the lock on `wg.Wait`, and allows the function to end.

Future

Future provides a placeholder for a value that's not yet known.

Applicability

Futures (also known as Promises or Delays[4]) are a synchronization construct that provide a placeholder for a value that's still being generated by an asynchronous process.

4 While these terms are often used interchangeably, they can also have shades of meaning depending on their context. I know. Please don't write me any angry letters about this.

This pattern isn't used as frequently in Go as in some other languages because channels can be often used in a similar way. For example, the long-running blocking function `BlockingInverse` (not shown) can be executed in a goroutine that returns the result (when it arrives) along a channel. The `ConcurrentInverse` function does exactly that, returning a channel that can be read when a result is available:

```
func ConcurrentInverse(m Matrix) <-chan Matrix {
    out := make(chan Matrix)

    go func() {
        out <- BlockingInverse(m)
        close(out)
    }()

    return out
}
```

Using `ConcurrentInverse`, one could then build a function to calculate the inverse product of two matrices:

```
func InverseProduct(a, b Matrix) Matrix {
    inva := ConcurrentInverse(a)
    invb := ConcurrentInverse(b)

    return Product(<-inva, <-invb)
}
```

This doesn't seem so bad, but it comes with some baggage that makes it undesirable for something like a public API. First, the caller has be careful to call `ConcurrentInverse` with the correct timing. To see what I mean, take a close look at the following:

```
return Product(<-ConcurrentInverse(a), <-ConcurrentInverse(b))
```

See the problem? Since the computation isn't started until `ConcurrentInverse` is actually called, this construct would be effectively executed serially, requiring twice the runtime.

What's more, when using channels in this way, functions with more than one return value will usually assign a dedicated channel to each member of the return list, which can become awkward as the return list grows or when the values need to be read by more than one goroutine.

The Future pattern contains this complexity by encapsulating it in an API that provides the consumer with a simple interface whose method can be called normally, blocking all calling routines until all of its results are resolved. The interface that the value satisfies doesn't even have to be constructed specially for that purpose; any interface that's convenient for the consumer can be used.

Participants

This pattern includes the following participants:

Future
: The interface that is received by the consumer to retrieve the eventual result.

SlowFunction
: A wrapper function around some function to be asynchronously executed; provides *Future*.

InnerFuture
: Satisfies the *Future* interface; includes an attached method that contains the result access logic.

Implementation

The API presented to the consumer is fairly straightforward: the programmer calls *SlowFunction*, which returns a value that satisfies the *Future* interface. *Future* may be a bespoke interface, as in the following example, or it may be something more like an `io.Reader` that can be passed to its own functions.

In actuality, when *SlowFunction* is called, it executes the core function of interest as a goroutine. In doing so, it defines channels to capture the core function's output, which it wraps in *InnerFuture*.

InnerFuture has one or more methods that satisfy the *Future* interface, which retrieve the values returned by the core function from the channels, cache them, and return them. If the values aren't available on the channel, the request blocks. If they have already been retrieved, the cached values are returned.

Sample code

In this example, we use a `Future` interface that the `InnerFuture` will satisfy:

```
type Future interface {
    Result() (string, error)
}
```

The `InnerFuture` struct is used internally to provide the concurrent functionality. In this example, it satisfies the `Future` interface, but could just as easily choose to satisfy something like `io.Reader` by attaching a `Read` method, for example:

```
type InnerFuture struct {
    once sync.Once
    wg   sync.WaitGroup

    res   string
    err   error
    resCh <-chan string
```

```
    errCh <-chan error
}

func (f *InnerFuture) Result() (string, error) {
    f.once.Do(func() {
        f.wg.Add(1)
        defer f.wg.Done()
        f.res = <-f.resCh
        f.err = <-f.errCh
    })

    f.wg.Wait()

    return f.res, f.err
}
```

In this implementation, the struct itself contains a channel and a variable for each value returned by the `Result` method. When `Result` is first called, it attempts to read the results from the channels and send them back to the `InnerFuture` struct so that subsequent calls to `Result` can immediately return the cached values.

Note the use of `sync.Once` and `sync.WaitGroup`. The former does what it says on the tin: it ensures that the function that's passed to it is called exactly once. The `WaitGroup` is used to make this function call thread safe: any calls after the first will be blocked at `wg.Wait` until the channel reads are complete.

`SlowFunction` is a wrapper around the core functionality that you want to run concurrently. It has the job of creating the results channels, running the core function in a goroutine, and creating and returning the `Future` implementation (`InnerFuture`, in this example):

```
func SlowFunction(ctx context.Context) Future {
    resCh := make(chan string)
    errCh := make(chan error)

    go func() {
        select {
        case <-time.After(time.Second * 2):
            resCh <- "I slept for 2 seconds"
            errCh <- nil
        case <-ctx.Done():
            resCh <- ""
            errCh <- ctx.Err()
        }
    }()

    return &InnerFuture{resCh: resCh, errCh: errCh}
}
```

To make use of this pattern, you need only call the `SlowFunction` and use the returned `Future` as you would any other value:

```
func main() {
    ctx := context.Background()
    future := SlowFunction(ctx)

    res, err := future.Result()
    if err != nil {
        fmt.Println("error:", err)
        return
    }

    fmt.Println(res)
}
```

This approach provides a reasonably good user experience. The programmer can create a `Future` and access it as they wish, and can even apply timeouts or deadlines with a `Context`.

Sharding

Sharding splits a large data structure into multiple partitions to localize the effects of read/write locks.

Applicability

The term *sharding* is typically used in the context of distributed state to describe data that is partitioned between server instances. This kind of *horizontal sharding* is commonly used by databases and other data stores to distribute load and provide redundancy.

A slightly different issue can sometimes affect highly concurrent services that have a shared data structure with a locking mechanism to protect it from conflicting writes. In this scenario, the locks that serve to ensure the fidelity of the data can also create a bottleneck when processes start to spend more time waiting for locks than they do doing their jobs. This unfortunate phenomenon is called *lock contention*.

While this might be resolved in some cases by scaling the number of instances, this also increases complexity and latency, because distributed locks need to be established, and writes need to establish consistency. An alternative strategy for reducing lock contention around shared data structures within an instance of a service is *vertical sharding*, in which a large data structure is partitioned into two or more structures, each representing a part of the whole. Using this strategy, only a portion of the overall structure needs to be locked at a time, decreasing overall lock contention.

Horizontal vs. Vertical Sharding

Large data structures can be sharded, or partitioned, in two different ways:

- *Horizontal sharding* is the partitioning of data across service instances. This can provide data redundancy and allow load to be balanced between instances, but also adds the latency and complexity that comes with distributed data.
- *Vertical sharding* is the partitioning of data within a single instance. This can reduce read/write contention between concurrent processes, but also doesn't scale or provide any redundancy.

Participants

This pattern includes the following participants:

ShardedMap
: An abstraction around one or more *Shards* providing read and write access as if the *Shards* were a single map.

Shard
: An individually lockable collection representing a single data partition.

Implementation

While idiomatic Go strongly prefers the use of memory sharing via channels over using locks to protect shared resources,[5] this isn't always possible. Maps are particularly unsafe for concurrent use, making the use of locks as a synchronization mechanism a necessary evil. Fortunately, Go provides `sync.RWMutex` for precisely this purpose.

`RWMutex` provides methods to establish both read and write locks, as demonstrated in the following. Using this method, any number of processes can establish simultaneous read locks as long as there are no open write locks; a process can establish a write lock only when there are no existing read or write locks. Attempts to establish additional locks will block until any locks ahead of it are released:

```
var items = struct{                                  // Struct with a map and a
    sync.RWMutex                                     // composed sync.RWMutex
    m map[string]int
}{m: make(map[string]int)}

func ThreadSafeRead(key string) int {
```

5 See the article, "Share Memory By Communicating," on *The Go Blog*.

```
    items.RLock()                                   // Establish read lock
    value := items.m[key]
    items.RUnlock()                                 // Release read lock
    return value
}

func ThreadSafeWrite(key string, value int) {
    items.Lock()                                    // Establish write lock
    items.m[key] = value
    items.Unlock()                                  // Release write lock
}
```

This strategy generally works perfectly fine. However, because locks allow access to only one process at a time, the average amount of time spent waiting for locks to clear in a read/write intensive application can increase dramatically with the number of concurrent processes acting on the resource. The resulting lock contention can potentially bottleneck key functionality.

Vertical sharding reduces lock contention by splitting the underlying data structure—usually a map—into several individually lockable maps. An abstraction layer provides access to the underlying shards as if they were a single structure (see Figure 4-5).

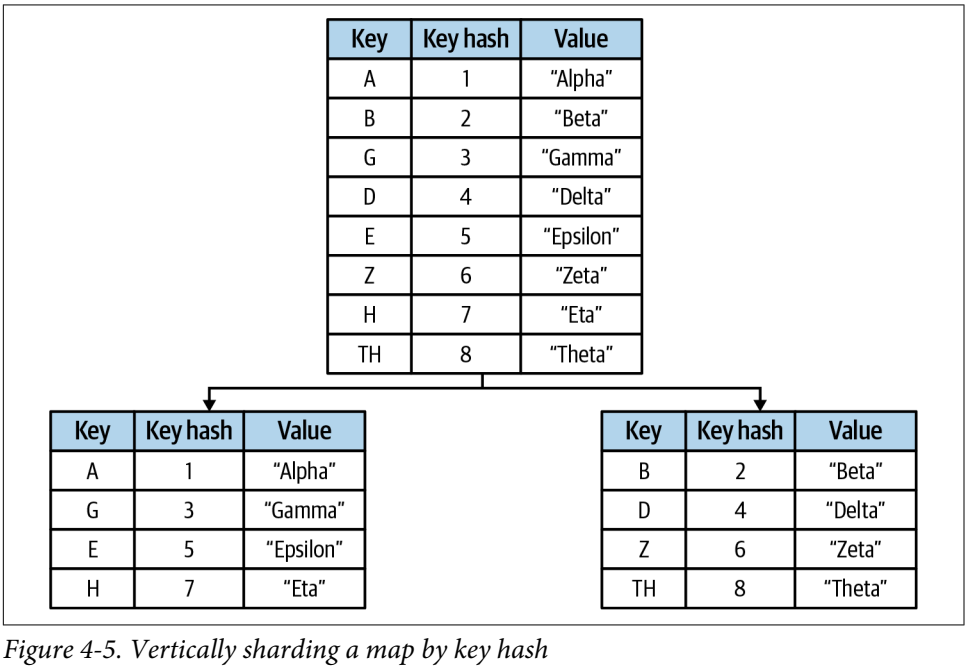

Figure 4-5. Vertically sharding a map by key hash

Internally, this is accomplished by creating an abstraction layer around what is essentially a map of maps. Whenever a value is read or written to the map abstraction, a

hash value is calculated for the key, which is then modded by the number of shards to generate a shard index. This allows the map abstraction to isolate the necessary locking to only the shard at that index.

Sample code

In the following example, we use the standard `sync` and `crypto/sha1` packages to implement a basic sharded map: `ShardedMap`.

Internally, `ShardedMap` is just a slice of pointers to some number of `Shard` values, but we define it as a type so we can attach methods to it. Each `Shard` includes a `map[string]interface{}` that contains that shard's data, and a composed `sync.RWMutex` so that it can be individually locked:

```
type Shard struct {
    sync.RWMutex                    // Compose from sync.RWMutex
    m map[string]interface{}        // m contains the shard's data
}

type ShardedMap []*Shard            // ShardedMap is a *Shards slice
```

Go doesn't have any concept of constructors, so we provide a `NewShardedMap` function to retrieve a new `ShardedMap`:

```
func NewShardedMap(nshards int) ShardedMap {
    shards := make([]*Shard, nshards)       // Initialize a *Shards slice

    for i := 0; i < nshards; i++ {
        shard := make(map[string]interface{})
        shards[i] = &Shard{m: shard}
    }

    return shards                           // A ShardedMap IS a *Shards slice!
}
```

`ShardedMap` has two unexported methods, `getShardIndex` and `getShard`, which are used to calculate a key's shard index and retrieve a key's correct shard, respectively. These could be easily combined into a single method, but slitting them this way makes them easier to test:

```
func (m ShardedMap) getShardIndex(key string) int {
    checksum := sha1.Sum([]byte(key))   // Use Sum from "crypto/sha1"
    hash := int(checksum[17])           // Pick an arbitrary byte as the hash
    return hash % len(m)                // Mod by len(m) to get index
}

func (m ShardedMap) getShard(key string) *Shard {
    index := m.getShardIndex(key)
    return m[index]
}
```

Note that the previous example has an obvious weakness: because it's effectively using a `byte`-sized value as the hash value, it can only handle up to 255 shards. If for some reason you want more than that, you can sprinkle some binary arithmetic on it: `hash := int(sum[13]) << 8 | int(sum[17])`.

Finally, we add methods to `ShardedMap` to allow a user to read and write values. Obviously these don't demonstrate all of the functionality a map might need. The source for this example is in the GitHub repository associated with this book, however, so please feel free to implement them as an exercise. A `Delete` and a `Contains` method would be nice:

```
func (m ShardedMap) Get(key string) interface{} {
    shard := m.getShard(key)
    shard.RLock()
    defer shard.RUnlock()

    return shard.m[key]
}

func (m ShardedMap) Set(key string, value interface{}) {
    shard := m.getShard(key)
    shard.Lock()
    defer shard.Unlock()

    shard.m[key] = value
}
```

When you do need to establish locks on all of the tables, it's generally best to do so concurrently. In the following, we implement a `Keys` function using goroutines and our old friend `sync.WaitGroup`:

```
func (m ShardedMap) Keys() []string {
    keys := make([]string, 0)              // Create an empty keys slice

    mutex := sync.Mutex{}                  // Mutex for write safety to keys

    wg := sync.WaitGroup{}                 // Create a wait group and add a
    wg.Add(len(m))                         // wait value for each slice

    for _, shard := range m {              // Run a goroutine for each slice
        go func(s *Shard) {
            s.RLock()                      // Establish a read lock on s

            for key := range s.m {         // Get the slice's keys
                mutex.Lock()
                keys = append(keys, key)
                mutex.Unlock()
            }

            s.RUnlock()                    // Release the read lock
            wg.Done()                      // Tell the WaitGroup it's done
```

```
            }(shard)
        }

        wg.Wait()                               // Block until all reads are done

        return keys                             // Return combined keys slice
    }
```

Using `ShardedMap` isn't quite like using a standard map unfortunately, but while it's different, it's no more complicated:

```
    func main() {
        shardedMap := NewShardedMap(5)

        shardedMap.Set("alpha", 1)
        shardedMap.Set("beta", 2)
        shardedMap.Set("gamma", 3)

        fmt.Println(shardedMap.Get("alpha"))
        fmt.Println(shardedMap.Get("beta"))
        fmt.Println(shardedMap.Get("gamma"))

        keys := shardedMap.Keys()
        for _, k := range keys {
            fmt.Println(k)
        }
    }
```

Perhaps the greatest downside of the `ShardedMap` (besides its complexity, of course) is the loss of type safety associated with the use of `interface{}`, and the subsequent requirement of type assertions. Hopefully, with the impending release of generics for Go, this will soon be (or perhaps already is, depending on when you read this) a problem of the past!

Summary

This chapter covered quite a few very interesting—and useful—idioms. There are probably many more,[6] but these are the ones I felt were most important, either because they're somehow practical in a directly applicable way, or because they showcase some interesting feature of the Go language. Often both.

In Chapter 5 we'll move on to the next level, taking some of the things we discussed in Chapters 3 and 4, and putting them into practice by building a simple key-value store from scratch!

6 Did I leave out your favorite? Let me know, and I'll try to include it in the next edition!

CHAPTER 5

Building a Cloud Native Service

> Life was simple before World War II. After that, we had systems.[1]
>
> —Grace Hopper, *OCLC Newsletter (1987)*

In this chapter, our real work finally begins.

We'll weave together many of the materials discussed throughout Part II to create a service that will serve as the jumping-off point for the remainder of the book. As we go forward, we'll iterate on what we begin here, adding layers of functionality with each chapter until, at the conclusion, we have ourselves a true cloud native application.

Naturally, it won't be "production ready"—it will be missing important security features, for example—but it will provide a solid foundation for us to build upon.

But what do we build?

Let's Build a Service!

Okay. So. We need something to build.

It should be conceptually simple, straightforward enough to implement in its most basic form, but non-trivial and amenable to scaling and distributing. Something that we can iteratively refine over the remainder of the book. I put a lot of thought into this, considering different ideas for what our application would be, but in the end the answer was obvious.

We'll build ourselves a distributed key-value store.

1 Schieber, Philip. "The Wit and Wisdom of Grace Hopper." *OCLC Newsletter*, March/April, 1987, No. 167.

What's a Key-Value Store?

A key-value store is a kind of nonrelational database that stores data as a collection of key-value pairs. They're very different from the better-known relational databases, like Microsoft SQL Server or PostgreSQL, that we know and love.[2] Where relational databases structure their data among fixed tables with well-defined data types, key-value stores are far simpler, allowing users to associate a unique identifier (the key) with an arbitrary value.

In other words, at its heart, a key-value store is really just a map with a service endpoint, as shown in Figure 5-1. They're the simplest possible database.

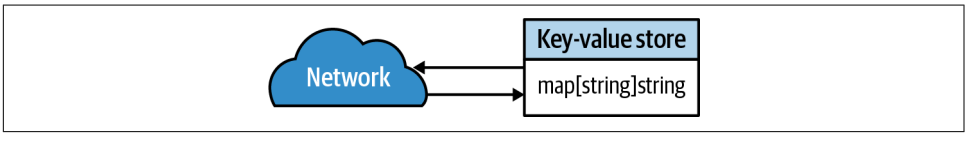

Figure 5-1. A key-value store is essentially a map with a service endpoint

Requirements

By the end of this chapter, we're going to have built a simple, nondistributed key-value store that can do all of the things that a (monolithic) key-value store should do.

- It must be able to store arbitrary key-value pairs.
- It must provide service endpoints that allow a user to put, get, and delete key-value pairs.
- It must be able to persistently store its data in some fashion.

Finally, we'd like the service to be idempotent. But why?

What Is Idempotence and Why Does It Matter?

The concept of *idempotence* has its origins in algebra, where it describes particular properties of certain mathematical operations. Fortunately, this isn't a math book. We're not going to talk about that (except in the sidebar at the end of this section).

In the programming world, an operation (such as a method or service call) is idempotent if calling it once has the same effect as calling it multiple times. For example, the assignment operation `x=1` is idempotent, because `x` will always be `1` no matter how many times you assign it. Similarly, an HTTP `PUT` method is idempotent because `PUT`-ting a resource in a place multiple times won't change anything: it won't get any

2 For some definition of "love."

more PUT the second time.[3] The operation x+=1, however, is not idempotent, because every time that it's called, a new state is produced.

Less discussed, but also important, is the related property of *nullipotence*, in which a function or operation has no side effect at all. For example, the x=1 assignment and an HTTP PUT are idempotent but not nullipotent because they trigger state changes. Assigning a value to itself, such as x=x, is nullipotent because no state has changed as a result of it. Similarly, simply reading data, as with an HTTP GET, usually has no side effects, so it's also nullipotent.

Of course, that's all very nice in theory, but why should we care in the real world? Well, as it turns out, designing your service methods to be idempotent provides a number of very real benefits:

Idempotent operations are safer
: What if you make a request to a service, but get no response? You'll probably try again. But what if it heard you the first time?[4] If the service method is idempotent, then no harm done. But if it's not, you could have a problem. This scenario is more common than you think. Networks are unreliable. Responses can be delayed; packets can get dropped.

Idempotent operations are often simpler
: Idempotent operations are more self-contained and easier to implement. Compare, for example, an idempotent PUT method that simply adds a key-value pair into a backing data store, and a similar but nonidempotent CREATE method that returns an error if the data store already contains the key. The PUT logic is simple: receive request, set value. The CREATE, on the other hand, requires additional layers of error checking and handling, and possibly even distributed locking and coordination among any service replicas, making its service harder to scale.

Idempotent operations are more declarative
: Building an idempotent API encourages the designer to focus on end-states, encouraging the production of methods that are more *declarative*: they allow users to tell a service *what needs to be done*, instead of telling it *how to do it*. This may seem to be a fine point, but declarative methods—as opposed to *imperative methods*—free users from having to deal with low-level constructs, allowing them to focus on their goals and minimizing potential side-effects.

In fact, idempotence provides such an advantage, particularly in a cloud native context, that some very smart people have even gone so far as to assert that it's a

3 If it does, something is very wrong.

4 Or, like my son, was only *pretending* not to hear you.

synonym for "cloud native."[5] I don't think that I'd go quite that far, but I *would* say that if your service aims to be cloud native, accepting any less than idempotence is asking for trouble.

The Mathematical Definition of Idempotence

The origin of idempotence is in mathematics, where it describes an operation that can be applied multiple times without changing the result beyond the initial application.

In purely mathematical terms: a function is idempotent if $f(f(x)) = f(x)$ for all x.

For example, taking the absolute value $abs(x)$ of an integer number x is an idempotent function because $abs(x) = abs(abs(x))$ is true for each real number x.

The Eventual Goal

These requirements are quite a lot to chew on, but they represent the absolute minimum for our key-value store to be usable. Later on we'll add some important basic functionality, like support for multiple users and data encryption in transit. More importantly, though, we'll introduce techniques and technologies that make the service more scalable, resilient, and generally capable of surviving and thriving in a cruel, uncertain universe.

Generation 0: The Core Functionality

Okay, let's get started. First things first. Without worrying about user requests and persistence, let's first build the core functions, which can be called later from whatever web framework we decide to use.

Storing arbitrary key-value pairs
: For now, we can implement this with a simple map, but what kind? For the sake of simplicity, we'll limit ourselves to keys and values that are simple strings, though we may choose to allow arbitrary types later. We'll just use a simple `map[string]string` as our core data structure.

Allow put, get, and delete of key-value pairs
: In this initial iteration, we'll create a simple Go API that we can call to perform the basic modification operations. Partitioning the functionality in this way will make it easier to test and easier to update in future iterations.

5 "Cloud native is not a synonym for microservices… if cloud native has to be a synonym for anything, it would be idempotent, which definitely needs a synonym." —Holly Cummins (Cloud Native London 2018).

Your Super Simple API

The first thing that we need to do is to create our map. The heart of our key-value store:

```
var store = make(map[string]string)
```

Isn't it a beauty? So simple. Don't worry, we'll make it more complicated later.

The first function that we'll create is, appropriately, `PUT`, which will be used to add records to the store. It does exactly what its name suggests: it accepts `key` and `value` strings, and puts them into `store`. `PUT`'s function signature includes an `error` return, which we'll need later:

```
func Put(key string, value string) error {
    store[key] = value

    return nil
}
```

Because we're making the conscious choice to create an idempotent service, `Put` doesn't check to see whether an existing key-value pair is being overwritten, so it will happily do so if asked. Multiple executions of `Put` with the same parameters will have the same result, regardless of any current state.

Now that we've established a basic pattern, writing the `Get` and `Delete` operations is just a matter of following through:

```
var ErrorNoSuchKey = errors.New("no such key")

func Get(key string) (string, error) {
    value, ok := store[key]

    if !ok {
        return "", ErrorNoSuchKey
    }

    return value, nil
}

func Delete(key string) error {
    delete(store, key)

    return nil
}
```

But look carefully: see how when `Get` returns an error, it doesn't use `errors.New`? Instead it returns the prebuilt `ErrorNoSuchKey` error value. But why? This is an example of a *sentinel error*, which allows the consuming service to determine exactly what type of error it's receiving and to respond accordingly. For example, it might do something like this:

```
    if errors.Is(err, ErrorNoSuchKey) {
        http.Error(w, err.Error(), http.StatusNotFound)
        return
    }
```

Now that you have your absolute minimal function set (really, really minimal), don't forget to write tests. We're not going to do that here, but if you're feeling anxious to move forward (or lazy—lazy works too) you can grab the code from the GitHub repository created for this book.

Generation 1: The Monolith

Now that we have a minimally functional key-value API, we can begin building a service around it. We have a few different options for how to do this. We could use something like GraphQL. There are some decent third-party packages out there that we could use, but we don't have the kind of complex data landscape to necessitate it. We could also use remote procedure call (RPC), which is supported by the standard `net/rpc` package, or even gRPC, but these require additional overhead for the client, and again our data just isn't complex enough to warrant it.

That leaves us with representational state transfer (REST). REST isn't a lot of people's favorite, but it *is* simple, and it's perfectly adequate for our needs.

Building an HTTP Server with net/http

Go doesn't have any web frameworks that are as sophisticated or historied as something like Django or Flask. What it does have, however, is a strong set of standard libraries that are perfectly adequate for 80% of use cases. Even better: they're designed to be extensible, so there *are* a number of Go web frameworks that extend them.

For now, let's take a look at the standard HTTP handler idiom in Go, in the form of a "Hello World" as implemented with `net/http`:

```
package main

import (
    "log"
    "net/http"
)

func helloGoHandler(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello net/http!\n"))
}

func main() {
    http.HandleFunc("/", helloGoHandler)

    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

In the previous example, we define a method, `helloGoHandler`, which satisfies the definition of a `http.HandlerFunc`:

```
type HandlerFunc func(http.ResponseWriter, *http.Request)
```

The `http.ResponseWriter` and a `*http.Request` parameters can be used to construct the HTTP response and retrieve the request, respectively. You can use the `http.HandleFunc` function to register `helloGoHandler` as the handler function for any request that matches a given pattern (the root path, in this example).

Once you've registered our handlers, you can call `ListenAndServe`, which listens on the address `addr`. It also accepts a second parameter, set to `nil` in our example.

You'll notice that `ListenAndServe` is also wrapped in a `log.Fatal` call. This is because `ListenAndServe` always stops the execution flow, only returning in the event of an error. Therefore, it always returns a non-nil `error`, which we always want to log.

The previous example is a complete program that can be compiled and run using `go run`:

```
$ go run .
```

Congratulations! You're now running the world's tiniest web service. Now go ahead and test it with `curl` or your favorite web browser:

```
$ curl http://localhost:8080
Hello net/http!
```

ListenAndServe, Handlers, and HTTP Request Multiplexers

The `http.ListenAndServe` function starts an HTTP server with a given address and handler. If the handler is `nil`, which it usually is when you're using only the standard `net/http` library, the `DefaultServeMux` value is used. But what's a handler? What is `DefaultServeMux`? *What's a "mux"?*

A `Handler` is any type that satisfies the `Handler` interface by providing a `ServeHTTP` method, defined in the following:

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

Most handler implementations, including the default handler, act as a "mux"—short for "multiplexer"—that can direct incoming signals to one of several possible outputs. When a request is received by a service that's been started by `ListenAndServe`, it's the job of a mux to compare the requested URL to the registered patterns and call the handler function associated with the one that matches most closely.

`DefaultServeMux` is a global value of type `ServeMux`, which implements the default HTTP multiplexer logic.

Building an HTTP Server with gorilla/mux

For many web services the `net/http` and `DefaultServeMux` will be perfectly sufficient. However, sometimes you'll need the additional functionality provided by a third-party web toolkit. A popular choice is Gorilla, which, while being relatively new and less fully developed and resource-rich than something like Django or Flask, does build on Go's standard `net/http` package to provide some excellent enhancements.

The `gorilla/mux` package—one of several packages provided as part of the Gorilla web toolkit—provides an HTTP request router and dispatcher that can fully replace `DefaultServeMux`, Go's default service handler, to add several very useful enhancements to request routing and handling. We're not going to make use of these features just yet, but they will come in handy going forward. If you're curious and/or impatient, however, you can take a look at the `gorilla/mux` documentation for more information.

Creating a minimal service

Once you've done so, making use of the minimal `gorilla/mux` router is a matter of adding an import and one line of code: the initialization of a new router, which can be passed to the `handler` parameter of `ListenAndServe`:

```
package main

import (
    "log"
    "net/http"

    "github.com/gorilla/mux"
)

func helloMuxHandler(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello gorilla/mux!\n"))
}

func main() {
    r := mux.NewRouter()

    r.HandleFunc("/", helloMuxHandler)

    log.Fatal(http.ListenAndServe(":8080", r))
}
```

So you should be able to just run this now with `go run`, right? Give it a try:

```
$ go run .
main.go:7:5: cannot find package "github.com/gorilla/mux" in any of:
        /go/1.15.8/libexec/src/github.com/gorilla/mux (from $GOROOT)
        /go/src/github.com/gorilla/mux (from $GOPATH)
```

It turns out that you can't (yet). Since you're now using a third-party package—a package that lives outside the standard library—you're going to have to use Go modules.

Initializing your project with Go modules

Using a package from outside the standard library requires that you make use of Go modules, which were introduced in Go 1.12 to replace an essentially nonexistent dependency management system with one that's explicit and actually quite painless to use. All of the operations that you'll use for managing your dependencies will use one of a small handful of `go mod` commands.

The first thing you're going to have to do is initialize your project. Start by creating a new, empty directory, `cd` into it, and create (or move) the Go file for your service there. Your directory should now contain only a single Go file.

Next, use the `go mod init` command to initialize the project. Typically, if a project will be imported by other projects, it'll have to be initialized with its import path. This is less important for a standalone service like ours, though, so you can be a little more lax about the name you choose. I'll just use `example.com/gorilla`; you can use whatever you like:

```
$ go mod init example.com/gorilla
go: creating new go.mod: module example.com/gorilla
```

You'll now have an (almost) empty module file, `go.mod`, in your directory:[6]

```
$ cat go.mod
module example.com/gorilla

go 1.15
```

Next, we'll want to add our dependencies, which can be done automatically using `go mod tidy`:

```
$ go mod tidy
go: finding module for package github.com/gorilla/mux
go: found github.com/gorilla/mux in github.com/gorilla/mux v1.8.0
```

If you check your `go.mod` file, you'll see that the dependency (and a version number) have been added:

6 Isn't this exciting?

```
$ cat go.mod
module example.com/gorilla

go 1.15

require github.com/gorilla/mux v1.8.0
```

Believe it or not, that's all you need. If your required dependencies change in the future you need only run `go mod tidy` again to rebuild the file. Now try again to start your service:

```
$ go run .
```

Since the service runs in the foreground, your terminal should pause. Calling the endpoint with `curl` from another terminal or browsing to it with a browser should provide the expected response:

```
$ curl http://localhost:8080
Hello gorilla/mux!
```

Success! But surely you want your service to do more than print a simple string, right? Of course you do. Read on!

Variables in URI paths

The Gorilla web toolkit provides a wealth of additional functionality over the standard `net/http` package, but one feature is particularly interesting right now: the ability to create paths with variable segments, which can even optionally contain a regular expression pattern. Using the `gorilla/mux` package, a programmer can define variables using the format `{name}` or `{name:pattern}`, as follows:

```
r := mux.NewRouter()
r.HandleFunc("/products/{key}", ProductHandler)
r.HandleFunc("/articles/{category}/", ArticlesCategoryHandler)
r.HandleFunc("/articles/{category}/{id:[0-9]+}", ArticleHandler)
```

The `mux.Vars` function conveniently allows the handler function to retrieve the variable names and values as a `map[string]string`:

```
vars := mux.Vars(request)
category := vars["category"]
```

In the next section we'll use this ability to allow clients to perform operations on arbitrary keys.

So many matchers

Another feature provided by `gorilla/mux` is that it allows a variety of *matchers* to be added to routes that let the programmer add a variety of additional matching request criteria. These include (but aren't limited to) specific domains or subdomains, path

prefixes, schemes, headers, and even custom matching functions of your own creation.

Matchers can be applied by calling the appropriate function on the `*Route` value that's returned by Gorilla's `HandleFunc` implementation. Each matcher function returns the affected `*Route`, so they can be chained. For example:

```
r := mux.NewRouter()

r.HandleFunc("/products", ProductsHandler).
    Host("www.example.com").              // Only match a specific domain
    Methods("GET", "PUT").                // Only match GET+PUT methods
    Schemes("http")                       // Only match the http scheme
```

See the `gorilla/mux` documentation for an exhaustive list of available matcher functions.

Building a RESTful Service

Now that you know how to use Go's standard HTTP library, you can use it to create a RESTful service that a client can interact with to execute call to the API you built in "Your Super Simple API" on page 111. Once you've done this you'll have implemented the absolute minimal viable key-value store.

Your RESTful methods

We're going to do our best to follow RESTful conventions, so our API will consider every key-value pair to be a distinct resource with a distinct URI that can be operated upon using the various HTTP methods. Each of our three basic operations—Put, Get, and Delete—will be requested using a different HTTP method that we summarize in Table 5-1.

The URI for your key-value pair resources will have the form `/v1/key/{key}`, where `{key}` is the unique key string. The `v1` segment indicates the API version. This convention is often used to manage API changes, and while this practice is by no means required or universal, it can be helpful for managing the impact of future changes that could break existing client integrations.

Table 5-1. Your RESTful methods

Functionality	Method	Possible Statuses
Put a key-value pair into the store	`PUT`	`201 (Created)`
Read a key-value pair from the store	`GET`	`200 (OK), 404 (Not Found)`
Delete a key-value pair	`DELETE`	`200 (OK)`

In "Variables in URI paths" on page 116, we discussed how to use the `gorilla/mux` package to register paths that contain variable segments, which will allow you to define a single variable path that handles *all* keys, mercifully freeing you from having to register every key independently. Then, in "So many matchers" on page 116, we discussed how to use route matchers to direct requests to specific handler functions based on various nonpath criteria, which you can use to create a separate handler function for each of the five HTTP methods that you'll be supporting.

Implementing the create function

Okay, you now have everything you need to get started! So, let's go ahead and implement the handler function for the creation of key-value pairs. This function has to be sure to satisfy several requirements:

- It must only match `PUT` requests for `/v1/key/{key}`.
- It must call the `Put` method from "Your Super Simple API" on page 111.
- It must respond with a `201 (Created)` when a key-value pair is created.
- It must respond to unexpected errors with a `500 (Internal Server Error)`.

All of the previous requirements are implemented in the `keyValuePutHandler` function. Note how the key's value is retrieved from the request body:

```
// keyValuePutHandler expects to be called with a PUT request for
// the "/v1/key/{key}" resource.
func keyValuePutHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)                    // Retrieve "key" from the request
    key := vars["key"]

    value, err := io.ReadAll(r.Body)       // The request body has our value
    defer r.Body.Close()

    if err != nil {                        // If we have an error, report it
        http.Error(w,
            err.Error(),
            http.StatusInternalServerError)
        return
    }
```

```
    err = Put(key, string(value))          // Store the value as a string
    if err != nil {                        // If we have an error, report it
        http.Error(w,
            err.Error(),
            http.StatusInternalServerError)
        return
    }

    w.WriteHeader(http.StatusCreated)      // All good! Return StatusCreated
}
```

Now that you have your "key-value create" handler function, you can register it with your Gorilla request router for the desired path and method:

```
func main() {
    r := mux.NewRouter()

    // Register keyValuePutHandler as the handler function for PUT
    // requests matching "/v1/{key}"
    r.HandleFunc("/v1/{key}", keyValuePutHandler).Methods("PUT")

    log.Fatal(http.ListenAndServe(":8080", r))
}
```

Now that you have your service put together, you can run it using `go run .` from the project root. Do that now, and send it some requests to see how it responds.

First, use our old friend `curl` to send a `PUT` containing a short snippet of text to the `/v1/key-a` endpoint to create a key named `key-a` with a value of `Hello, key-value store!`:

```
$ curl -X PUT -d 'Hello, key-value store!' -v http://localhost:8080/v1/key-a
```

Executing this command provides the following output. The complete output was quite wordy, so I've selected the relevant bits for readability:

```
> PUT /v1/key-a HTTP/1.1
< HTTP/1.1 201 Created
```

The first portion, prefixed with a greater-than symbol (`>`), shows some details about the request. The last portion, prefixed with a less-than symbol (`<`), gives details about the server response.

In this output you can see that you did in fact transmit a `PUT` to the `/v1/key-a` endpoint, and that the server responded with a `201 Created`—as expected.

What if you hit the `/v1/key-a` endpoint with an unsupported `GET` method? Assuming that the matcher function is working correctly, you should receive an error message:

```
$ curl -X GET -v http://localhost:8080/v1/key-a
> GET /v1/key-a HTTP/1.1
< HTTP/1.1 405 Method Not Allowed
```

Indeed, the server responds with a `405 Method Not Allowed` error. Everything seems to be working correctly.

Implementing the read function

Now that your service has a fully functioning `Put` method, it sure would be nice if you could read your data back! For our next trick, we're going to implement the `Get` functionality, which has the following requirements:

- It must only match `GET` requests for `/v1/key/{key}`.
- It must call the `Get` method from "Your Super Simple API" on page 111.
- It must respond with a `404 (Not Found)` when a requested key doesn't exist.
- It must respond with the requested value and a status `200` if the key exists.
- It must respond to unexpected errors with a `500 (Internal Server Error)`.

All of the previous requirements are implemented in the `keyValueGetHandler` function. Note how the value is written to `w`—the handler function's `http.ResponseWriter` parameter—after it's retrieved from the key-value API:

```
func keyValueGetHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)                    // Retrieve "key" from the request
    key := vars["key"]

    value, err := Get(key)                 // Get value for key
    if errors.Is(err, ErrorNoSuchKey) {
        http.Error(w,err.Error(), http.StatusNotFound)
        return
    }
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }

    w.Write([]byte(value))                 // Write the value to the response
}
```

And now that you have the "get" handler function, you can register it with the request router alongside the "put" handler:

```
func main() {
    r := mux.NewRouter()

    r.HandleFunc("/v1/{key}", keyValuePutHandler).Methods("PUT")
    r.HandleFunc("/v1/{key}", keyValueGetHandler).Methods("GET")

    log.Fatal(http.ListenAndServe(":8080", r))
}
```

Now let's fire up your newly improved service and see if it works:

```
$ curl -X PUT -d 'Hello, key-value store!' -v http://localhost:8080/v1/key-a
> PUT /v1/key-a HTTP/1.1
< HTTP/1.1 201 Created

$ curl -v http://localhost:8080/v1/key-a
> GET /v1/key-a HTTP/1.1
< HTTP/1.1 200 OK
Hello, key-value store!
```

It works! Now that you can get your values back, you're able to test for idempotence as well. Let's repeat the requests and make sure that you get the same results:

```
$ curl -X PUT -d 'Hello, key-value store!' -v http://localhost:8080/v1/key-a
> PUT /v1/key-a HTTP/1.1
< HTTP/1.1 201 Created

$ curl -v http://localhost:8080/v1/key-a
> GET /v1/key-a HTTP/1.1
< HTTP/1.1 200 OK
Hello, key-value store!
```

You do! But what if you want to overwrite the key with a new value? Will the subsequent `GET` have the new value? You can test that by changing the value sent by your `curl` slightly to be `Hello, again, key-value store!`:

```
$ curl -X PUT -d 'Hello, again, key-value store!' \
    -v http://localhost:8080/v1/key-a
> PUT /v1/key-a HTTP/1.1
< HTTP/1.1 201 Created

$ curl -v http://localhost:8080/v1/key-a
> GET /v1/key-a HTTP/1.1
< HTTP/1.1 200 OK
Hello, again, key-value store!
```

As expected, the `GET` responds back with a `200` status and your new value.

Finally, to complete your method set you'll just need to create a handler for the `DELETE` method. I'll leave that as an exercise, though. Enjoy!

Making Your Data Structure Concurrency-Safe

Maps in Go are not atomic and are not safe for concurrent use. Unfortunately, you now have a service designed to handle concurrent requests that's wrapped around exactly such a map.

So what do you do? Well, typically when a programmer has a data structure that needs to be read from and written to by concurrently executing goroutines, they'll use something like a mutex—also known as a lock—to act as a synchronization

mechanism. By using a mutex in this way, you can ensure that exactly one process has exclusive access to a particular resource.

Fortunately, you don't need to implement this yourself:[7] Go's `sync` package provides exactly what you need in the form of `sync.RWMutex`. The following statement uses the magic of composition to create an *anonymous struct* that contains your map and an embedded `sync.RWMutex`:

```
var myMap = struct{
    sync.RWMutex
    m map[string]string
}{m: make(map[string]string)}
```

The `myMap` struct has all of the methods from the embedded `sync.RWMutex`, allowing you to use the `Lock` method to take the write lock when you want to write to the `myMap` map:

```
myMap.Lock()                                // Take a write lock
myMap.m["some_key"] = "some_value"
myMap.Unlock()                              // Release the write lock
```

If another process has either a read or write lock, then `Lock` will block until that lock is released.

Similarly, to read from the map, you use the `RLock` method to take the read lock:

```
myMap.RLock()                               // Take a read lock
value := myMap.m["some_key"]
myMap.RUnlock()                             // Release the read lock

fmt.Println("some_key:", value)
```

Read locks are less restrictive than write locks in that any number of processes can simultaneously take read locks. However, `RLock` will block until any open write locks are released.

Integrating a read-write mutex into your application

Now that you know how to use a `sync.RWMutex` to implement a basic read-write mutex, you can go back and work it into the code you created for "Your Super Simple API" on page 111.

First, you'll want to refactor the `store` map.[8] You can construct it like `myMap`, i.e., as an anonymous struct that contains the map and an embedded `sync.RWMutex`:

7 It's a good thing too. Mutexes can be pretty tedious to implement correctly!

8 Didn't I tell you that we'd make it more complicated?

```
var store = struct{
    sync.RWMutex
    m map[string]string
}{m: make(map[string]string)}
```

Now that you have your `store` struct, you can update the `Get` and `Put` functions to establish the appropriate locks. Because `Get` only needs to *read* the `store` map, it'll use `RLock` to take a read lock only. `Put`, on the other hand, needs to *modify* the map, so it'll need to use `Lock` to take a write lock:

```
func Get(key string) (string, error) {
    store.RLock()
    value, ok := store.m[key]
    store.RUnlock()

    if !ok {
        return "", ErrorNoSuchKey
    }

    return value, nil
}

func Put(key string, value string) error {
    store.Lock()
    store.m[key] = value
    store.Unlock()

    return nil
}
```

The pattern here is clear: if a function needs to modify the map (`Put`, `Delete`), it'll use `Lock` to take a write lock. If it only needs to read existing data (`Get`), it'll use `RLock` to take a read lock. We leave the creation of the `Delete` function as an exercise for the reader.

Don't forget to release your locks, and make sure you're releasing the correct lock type!

Generation 2: Persisting Resource State

One of the stickiest challenges with distributed cloud native applications is how to handle state.

There are various techniques for distributing the state of application resources between multiple service instances, but for now we're just going to concern ourselves

with the minimum viable product and consider two ways of maintaining the state of our application:

- In "Storing State in a Transaction Log File" on page 126, you'll use a file-based *transaction log* to maintain a record of every time a resource is modified. If a service crashes, is restarted, or otherwise finds itself in an inconsistent state, a transaction log allows a service to reconstruct original state simply by replaying the transactions.
- In "Storing State in an External Database" on page 137, you'll use an external database instead of a file to store a transaction log. It might seem redundant to use a database given the nature of the application you're building, but externalizing data into another service designed specifically for that purpose is a common means of sharing state between service replicas and providing resilience.

You may be wondering why you're using a transaction log strategy to record the events when you could just use the database to store the values themselves. This makes sense when you intend to store your data in memory most of the time, only accessing your persistence mechanism in the background and at startup time.

This also affords you another opportunity: given that you're creating two different implementations of a similar functionality—a transaction log written both to a file and to a database—you can describe your functionality with an interface that both implementations can satisfy. This could come in quite handy, especially if you want to be able to choose the implementation according to your needs.

Application State Versus Resource State

The term "stateless" is used a lot in the context of cloud native architecture, and state is often regarded as a Very Bad Thing. But what is state, exactly, and why is it so bad? Does an application have to be completely devoid of any kind of state to be "cloud native"? The answer is… well, it's complicated.

First, it's important to draw a distinction between *application state* and *resource state*. These are very different things, but they're easily confused.

Application state
: Server-side data about the application or how it's being used by a client. A common example is client session tracking, such as to associate them with their access credentials or some other application context.

Resource state
: The current state of a resource within a service at any point of time. It's the same for every client, and has nothing to do with the interaction between client and server.

Any state introduces technical challenges, but application state is particularly problematic because it forces services to depend on *server affinity*—sending each of a user's requests to the same server where their session was initiated—resulting in a more complex application and making it hard to destroy or replace service replicas.

State and statelessness will be discussed in quite a bit more detail in "State and Statelessness" on page 195.

What's a Transaction Log?

In its simplest form, a *transaction log* is just a log file that maintains a history of mutating changes executed by the data store. If a service crashes, is restarted, or otherwise finds itself in an inconsistent state, a transaction log makes it possible to replay the transactions to reconstruct the service's functional state.

Transaction logs are commonly used by database management systems to provide a degree of data resilience against crashes or hardware failures. However, while this technique can get quite sophisticated, we'll be keeping ours pretty straightforward.

Your transaction log format

Before we get to the code, let's decide what the transaction log should contain.

We'll assume that your transaction log will be read only when your service is restarted or otherwise needs to recover its state, and that it'll be read from top to bottom, sequentially replaying each event. It follows that your transaction log will consist of an ordered list of mutating events. For speed and simplicity, a transaction log is also generally append-only, so when a record is deleted from your key-value store, for example, a `delete` is recorded in the log.

Given everything we've discussed so far, each recorded transaction event will need to include the following attributes:

Sequence number
: A unique record ID, in monotonically increasing order.

Event type
: A descriptor of the type of action taken; this can be `PUT` or `DELETE`.

Key
: A string containing the key affected by this transaction.

Value
: If the event is a `PUT`, the value of the transaction.

Nice and simple. Hopefully we can keep it that way.

Your transaction logger interface

The first thing we're going to do is define a `TransactionLogger` interface. For now, we're only going to define two methods: `WritePut` and `WriteDelete`, which will be used to write `PUT` and `DELETE` events, respectively, to a transaction log:

```
type TransactionLogger interface {
    WriteDelete(key string)
    WritePut(key, value string)
}
```

You'll no doubt want to add other methods later, but we'll cross that bridge when we come to it. For now, let's focus on the first implementation and add additional methods to the interface as we come across them.

Storing State in a Transaction Log File

The first approach we'll take is to use the most basic (and most common) form of transaction log, which is just an append-only log file that maintains a history of mutating changes executed by the data store. This file-based implementation has some tempting pros, but some pretty significant cons as well:

Pros:

No downstream dependency
: There's no dependency on an external service that could fail or that we can lose access to.

Technically straightforward
: The logic isn't especially sophisticated. We can be up and running quickly.

Cons:

Harder to scale
: You'll need some additional way to distribute your state between nodes when you want to scale.

Uncontrolled growth
: These logs have to be stored on disk, so you can't let them grow forever. You'll need some way of compacting them.

Prototyping your transaction logger

Before we get to the code, let's make some design decisions. First, for simplicity, the log will be written in plain text; a binary, compressed format might be more time- and space-efficient, but we can always optimize later. Second, each entry will be written on its own line; this will make it much easier to read the data later.

Finally, each transaction will include the four fields listed in "Your transaction log format" on page 125, delimited by tabs. Once again, these are:

Sequence number
: A unique record ID, in monotonically increasing order.

Event type
: A descriptor of the type of action taken; this can be `PUT` or `DELETE`.

Key
: A string containing the key affected by this transaction.

Value
: If the event is a `PUT`, the value of the transaction.

Now that we've established these fundamentals, let's go ahead and define a type, `FileTransactionLogger`, which will implicitly implement the `TransactionLogger` interface described in "Your transaction logger interface" on page 126 by defining `WritePut` and `WriteDelete` methods for writing `PUT` and `DELETE` events, respectively, to the transaction log:

```
type FileTransactionLogger struct {
    // Something, something, fields
}

func (l *FileTransactionLogger) WritePut(key, value string) {
    // Something, something, logic
}

func (l *FileTransactionLogger) WriteDelete(key string) {
    // Something, something, logic
}
```

Clearly these methods are a little light on detail, but we'll flesh them out soon!

Defining the event type

Thinking ahead, we probably want the `WritePut` and `WriteDelete` methods to operate asynchronously. You could implement that using some kind of `events` channel that some concurrent goroutine could read from and perform the log writes. That sounds like a nice idea, but if you're going to do that you'll need some kind of internal representation of an "event."

That shouldn't give you too much trouble. Incorporating all of the fields that we listed in "Your transaction log format" on page 125 gives something like the `Event` struct, in the following:

```
type Event struct {
    Sequence  uint64                // A unique record ID
    EventType EventType             // The action taken
```

```
    Key       string                // The key affected by this transaction
    Value     string                // The value of a PUT the transaction
}
```

Seems straightforward, right? `Sequence` is the sequence number, and `Key` and `Value` are self-explanatory. But…what's an `EventType`? Well, it's whatever we say it is, and we're going to say that it's a constant that we can use to refer to the different types of events, which we've already established will include one each for `PUT` and `DELETE` events.

One way to do this might be to just assign some constant `byte` values, like this:

```
const (
    EventDelete byte = 1
    EventPut    byte = 2
)
```

Sure, this would work, but Go actually provides a better (and more idiomatic) way: `iota`. `iota` is a predefined value that can be used in a `constant` declaration to construct a series of related constant values.

Declaring Constants with Iota

When used in a `constant` declaration, `iota` represents successive untyped integer constants that can be used to construct a set of related constants. Its value restarts at zero in each `constant` declaration and increments with each constant assignment (whether or not the `iota` identifier is actually referenced).

An `iota` can also be operated upon. We demonstrate this in the following by using in multiplication, left binary shift, and division operations:

```
const (
    a = 42 * iota           // iota == 0; a == 0
    b = 1 << iota           // iota == 1; b == 2
    c = 3                   // iota == 2; c == 3 (iota increments anyway!)
    d = iota / 2            // iota == 3; d == 1
)
```

Because `iota` is itself an untyped number, you can use it to make typed assignments without explicit type casts. You can even assign `iota` to a `float64` value:

```
const (
    u         = iota * 42   // iota == 0; u == 0 (untyped integer constant)
    v float64 = iota * 42   // iota == 1; v == 42.0 (float64 constant)
)
```

The `iota` keyword allows implicit repetition, which makes it trivial to create arbitrarily long sets of related constants, like we do in the following with the numbers of bytes in various digital units:

```
type ByteSize uint64

const (
    _           = iota                    // iota == 0; ignore the zero value
    KB ByteSize = 1 << (10 * iota)        // iota == 1; KB == 2^10
    MB                                    // iota == 2; MB == 2^20
    GB                                    // iota == 3; GB == 2^30
    TB                                    // iota == 4; TB == 2^40
    PB                                    // iota == 5; PB == 2^50
)
```

Using the `iota` technique, you don't have to manually assign values to constants. Instead, you can do something like the following:

```
type EventType byte

const (
    _                     = iota          // iota == 0; ignore the zero value
    EventDelete EventType = iota          // iota == 1
    EventPut                              // iota == 2; implicitly repeat
)
```

This might not be a big deal when you only have two constants like we have here, but it can come in handy when you have a number of related constants and don't want to be bothered manually keeping track of which value is assigned to what.

If you're using `iota` as enumerations in serializations (as we are here), take care to only *append* to the list, and don't reorder or insert values in the middle, or you won't be able to deserialize later.

We now have an idea of what the `TransactionLogger` will look like, as well as the two primary write methods. We've also defined a struct that describes a single event, and created a new `EventType` type and used `iota` to define its legal values. Now we're finally ready to get started.

Implementing your FileTransactionLogger

We've made some progress. We know we want a `TransactionLogger` implementation with methods for writing events, and we've created a description of an event in code. But what about the `FileTransactionLogger` itself?

The service will want to keep track of the physical location of the transaction log, so it makes sense to have an `os.File` attribute representing that. It'll also need to remember the last sequence number that was assigned so it can correctly set each event's

sequence number; that can be kept as an unsigned 64-bit integer attribute. That's great, but how will the `FileTransactionLogger` actually write the events?

One possible approach would be to keep an `io.Writer` that the `WritePut` and `WriteDelete` methods can operate on directly, but that would be a single-threaded approach, so unless you explicitly execute them in goroutines, you may find yourself spending more time in I/O than you'd like. Alternatively, you could create a buffer from a slice of `Event` values that are processed by a separate goroutine. Definitely warmer, but too complicated.

After all, why go through all of that work when we can just use standard buffered channels? Taking our own advice, we end up with a `FileTransactionLogger` and `Write` methods that look like the following:

```
type FileTransactionLogger struct {
    events       chan<- Event       // Write-only channel for sending events
    errors       <-chan error       // Read-only channel for receiving errors
    lastSequence uint64             // The last used event sequence number
    file         *os.File           // The location of the transaction log
}

func (l *FileTransactionLogger) WritePut(key, value string) {
    l.events <- Event{EventType: EventPut, Key: key, Value: value}
}

func (l *FileTransactionLogger) WriteDelete(key string) {
    l.events <- Event{EventType: EventDelete, Key: key}
}

func (l *FileTransactionLogger) Err() <-chan error {
    return l.errors
}
```

You now have your `FileTransactionLogger`, which has a `uint64` value that's used to track the last-used event sequence number, a write-only channel that receives `Event` values, and `WritePut` and `WriteDelete` methods that send `Event` values into that channel.

But it looks like there might be a part left over: there's an `Err` method there that returns a receive-only error channel. There's a good reason for that. We've already mentioned that writes to the transaction log will be done concurrently by a goroutine that receives events from the `events` channel. While that makes for a more efficient write, it also means that `WritePut` and `WriteDelete` can't simply return an `error` when they encounter a problem, so we provide a dedicated error channel to communicate errors instead.

Creating a new FileTransactionLogger

If you've followed along so far you may have noticed that none of the attributes in the `FileTransactionLogger` have been initialized. If you don't fix this issue, it's going to cause some problems. Go doesn't have constructors, though, so to solve this you need to define a construction function, which you'll call, for lack of a better name,[9] `NewFileTransactionLogger`:

```
func NewFileTransactionLogger(filename string) (TransactionLogger, error) {
    file, err := os.OpenFile(filename, os.O_RDWR|os.O_APPEND|os.O_CREATE, 0755)
    if err != nil {
        return nil, fmt.Errorf("cannot open transaction log file: %w", err)
    }

    return &FileTransactionLogger{file: file}, nil
}
```

See how `NewFileTransactionLogger` returns a pointer type, but its return list specifies the decidedly nonpointy `TransactionLogger` interface type?

The reason for this is tricksy: while Go allows pointer types to implement an interface, it doesn't allow pointers to interface types.

`NewFileTransactionLogger` calls the `os.OpenFile` function to open the file specified by the `filename` parameter. You'll notice it accepts several flags that have been binary `OR`-ed together to set its behavior:

`os.O_RDWR`
: Opens the file in read/write mode.

`os.O_APPEND`
: Any writes to this file will append, not overwrite.

`os.O_CREATE`
: If the file doesn't exist, creates it.

There are quite a few of these flags besides the three we use here. Take a look at the `os` package documentation for a full listing.

We now have a construction function that ensures that the transaction log file is correctly created. But what about the channels? We *could* create the channels and spawn a goroutine with `NewFileTransactionLogger`, but that feels like we'd be adding too much mysterious functionality. Instead, we'll create a `Run` method.

9 That's a lie. There are probably lots of better names.

Appending entries to the transaction log

As of yet, there's nothing reading from the `events` channel, which is less than ideal. What's worse, the channels aren't even initialized. Let's change this by creating a `Run` method, shown in the following:

```
func (l *FileTransactionLogger) Run() {
    events := make(chan Event, 16)              // Make an events channel
    l.events = events

    errors := make(chan error, 1)               // Make an errors channel
    l.errors = errors

    go func() {
        for e := range events {                 // Retrieve the next Event

            l.lastSequence++                    // Increment sequence number

            _, err := fmt.Fprintf(              // Write the event to the log
                l.file,
                "%d\t%d\t%s\t%s\n",
                l.lastSequence, e.EventType, e.Key, e.Value)

            if err != nil {
                errors <- err
                return
            }
        }
    }()
}
```

This implementation is incredibly basic. It won't even correctly handle entries with whitespace or multiple lines!

The `Run` function does several important things.

First, it creates a buffered `events` channel. Using a buffered channel in our `TransactionLogger` means that calls to `WritePut` and `WriteDelete` won't block as long as the buffer isn't full. This lets the consuming service handle short bursts of events without being slowed by disk I/O. If the buffer does fill up, then the write methods will block until the log writing goroutine catches up.

Second, it creates an `errors` channel, which is also buffered, that we'll use to signal any errors that arise in the goroutine that's responsible for concurrently writing events to the transaction log. The buffer value of `1` allows us to send an error in a nonblocking manner.

Finally, it starts a goroutine that retrieves `Event` values from our `events` channel and uses the `fmt.Fprintf` function to write them to the transaction log. If `fmt.Fprintf` returns an `error`, the goroutine sends the error to the `errors` channel and halts.

Using a bufio.Scanner to play back file transaction logs

Even the best transaction log is useless if it's never read.[10] But how do we do that?

You'll need to read the log from the beginning and parse each line; `io.ReadString` and `fmt.Sscanf` let you do this with minimal fuss.

Channels, our dependable friends, will let your service stream the results to a consumer as it retrieves them. This might be starting to feel routine, but stop for a second to appreciate it. In most other languages the path of least resistance here would be to read in the entire file, stash it in an array, and finally loop over that array to replay the events. Go's convenient concurrency primitives make it almost trivially easy to stream the data to the consumer in a much more space- and memory-efficient way.

The `ReadEvents` method[11] demonstrates this:

```
func (l *FileTransactionLogger) ReadEvents() (<-chan Event, <-chan error) {
    scanner := bufio.NewScanner(l.file)      // Create a Scanner for l.file
    outEvent := make(chan Event)             // An unbuffered Event channel
    outError := make(chan error, 1)          // A buffered error channel

    go func() {
        var e Event

        defer close(outEvent)                // Close the channels when the
        defer close(outError)                // goroutine ends

        for scanner.Scan() {
            line := scanner.Text()

            if err := fmt.Sscanf(line, "%d\t%d\t%s\t%s",
                &e.Sequence, &e.EventType, &e.Key, &e.Value); err != nil {

                outError <- fmt.Errorf("input parse error: %w", err)
                return
            }

            // Sanity check! Are the sequence numbers in increasing order?
            if l.lastSequence >= e.Sequence {
                outError <- fmt.Errorf("transaction numbers out of sequence")
```

10 What makes a transaction log "good" anyway?

11 Naming is hard.

```
                return
            }

            l.lastSequence = e.Sequence     // Update last used sequence #

            outEvent <- e                   // Send the event along
        }

        if err := scanner.Err(); err != nil {
            outError <- fmt.Errorf("transaction log read failure: %w", err)
            return
        }
    }()

    return outEvent, outError
}
```

The `ReadEvents` method can really be said to be two functions in one: the outer function initializes the file reader, and creates and returns the event and error channels. The inner function runs concurrently to ingest the file contents line by line and send the results to the channels.

Interestingly, the `file` attribute of `TransactionLogger` is of type `*os.File`, which has a `Read` method that satisfies the `io.Reader` interface. `Read` is fairly low-level, but, if you wanted to, you could actually use it to retrieve the data. The `bufio` package, however, gives us a better way: the `Scanner` interface, which provides a convenient means for reading newline-delimited lines of text. We can get a new `Scanner` value by passing an `io.Reader`—an `os.File` in this case—to `bufio.NewScanner`.

Each call to the `scanner.Scan` method advances it to the next line, returning `false` if there aren't any lines left. A subsequent call to `scanner.Text` returns the line.

Note the `defer` statements in the inner anonymous goroutine. These ensure that the output channels are always closed. Because `defer` is scoped to the function they're declared in, these get called at the end of the goroutine, not `ReadEvents`.

You may recall from "Formatting I/O in Go" on page 34 that the `fmt.Sscanf` function provides a simple (but sometimes simplistic) means of parsing simple strings. Like the other methods in the `fmt` package, the expected format is specified using a format string with various "verbs" embedded: two digits (`%d`) and two strings (`%s`), separated by tab characters (`\t`). Conveniently, `fmt.Sscanf` lets you pass in pointers to the target values for each verb, which it can update directly.[12]

12 After all this time, I still think that's pretty neat.

Go's format strings have a long history dating back to C's `printf` and `scanf`, but they've been adopted by many other languages over the years, including C++, Java, Perl, PHP, Ruby, and Scala. You may already be familiar with them, but if you're not, take a break now to look at the `fmt` package documentation.

At the end of each loop the last-used sequence number is updated to the value that was just read, and the event is sent on its merry way. A minor point: note how the same `Event` value is reused on each iteration rather than creating a new one. This is possible because the `outEvent` channel is sending struct values, not *pointers* to struct values, so it already provides copies of whatever value we send into it.

Finally, the function checks for `Scanner` errors. The `Scan` method returns only a single boolean value, which is really convenient for looping. Instead, when it encounters an error, `Scan` returns `false` and exposes the error via the `Err` method.

Your transaction logger interface (redux)

Now that you've implemented a fully functional `FileTransactionLogger`, it's time to look back and see which of the new methods we can use to incorporate into the `TransactionLogger` interface. It actually looks like there are quite few we might like to keep in any implementation, leaving us with the following final form for the `TransactionLogger` interface:

```
type TransactionLogger interface {
    WriteDelete(key string)
    WritePut(key, value string)
    Err() <-chan error

    ReadEvents() (<-chan Event, <-chan error)

    Run()
}
```

Now that that's settled, you can finally start integrating the transaction log into your key-value service.

Initializing the FileTransactionLogger in your web service

The `FileTransactionLogger` is now complete! All that's left to do now is to integrate it with your web service. The first step of this is to add a new function that can create a new `TransactionLogger` value, read in and replay any existing events, and call `Run`.

First, let's add a `TransactionLogger` reference to our `service.go`. You can call it `logger` because naming is hard:

```
var logger TransactionLogger
```

Now that you have that detail out of the way, you can define your initialization method, which can look like the following:

```
func initializeTransactionLog() error {
    var err error

    logger, err = NewFileTransactionLogger("transaction.log")
    if err != nil {
        return fmt.Errorf("failed to create event logger: %w", err)
    }

    events, errors := logger.ReadEvents()
    e, ok := Event{}, true

    for ok && err == nil {
        select {
        case err, ok = <-errors:               // Retrieve any errors
        case e, ok = <-events:
            switch e.EventType {
            case EventDelete:                  // Got a DELETE event!
                err = Delete(e.Key)
            case EventPut:                     // Got a PUT event!
                err = Put(e.Key, e.Value)
            }
        }
    }

    logger.Run()

    return err
}
```

This function starts as you'd expect: it calls `NewFileTransactionLogger` and assigns it to `logger`.

The next part is more interesting: it calls `logger.ReadEvents`, and replays the results based on the `Event` values received from it. This is done by looping over a `select` with cases for both the `events` and `errors` channels. Note how the cases in the `select` use the format `case foo, ok = <-ch`. The `bool` returned by a channel read in this way will be `false` if the channel in question has been closed, setting the value of `ok` and terminating the `for` loop.

If we get an `Event` value from the `events` channel, we call either `Delete` or `Put` as appropriate; if we get an error from the `errors` channel, `err` will be set to a non-`nil` value and the `for` loop will be terminated.

Integrating FileTransactionLogger with your web service

Now that the initialization logic is put together, all that's left to do to complete the integration of the `TransactionLogger` is add exactly three function calls into the web

service. This is fairly straightforward, so we won't walk through it here. But, briefly, you'll need to add the following:

- `initializeTransactionLog` to the `main` method
- `logger.WriteDelete` to `keyValueDeleteHandler`
- `logger.WritePut` to `keyValuePutHandler`

We'll leave the actual integration as an exercise for the reader.[13]

Future improvements

We may have completed a minimal viable implementation of our transaction logger, but it still has plenty of issues and opportunities for improvement, such as:

- There aren't any tests.
- There's no `Close` method to gracefully close the file.
- The service can close with events still in the write buffer: events can get lost.
- Keys and values aren't encoded in the transaction log: multiple lines or white-space will fail to parse correctly.
- The sizes of keys and values are unbound: huge keys or values can be added, filling the disk.
- The transaction log is written in plain text: it will take up more disk space than it probably needs to.
- The log retains records of deleted values forever: it will grow indefinitely.

All of these would be impediments in production. I encourage you to take the time to consider—or even implement—solutions to one or more of these points.

Storing State in an External Database

Databases, and data, are at the core of many, if not most, business and web applications, so it makes perfect sense that Go includes a standard interface for SQL (or SQL-like) databases in its core libraries.

But does it make sense to use a SQL database to back our key-value store? After all, isn't it redundant for our data store to just depend on another data store? Yes, certainly. But externalizing a service's data into another service designed specifically for that purpose—a database—is a common pattern that allows state to be shared

13 You're welcome.

between service replicas and provides data resilience. Besides, the point is to show how you might interact with a database, not to design the perfect application.

In this section, you'll be implementing a transaction log backed by an external database and satisfying the `TransactionLogger` interface, just as you did in "Storing State in a Transaction Log File" on page 126. This would certainly work, and even have some benefits as mentioned previously, but it comes with some tradeoffs:

Pros:

Externalizes application state
: Less need to worry about distributed state and closer to "cloud native."

Easier to scale
: Not having to share data between replicas makes scaling out *easier* (but not *easy*).

Cons:

Introduces a bottleneck
: What if you had to scale way up? What if all replicas had to read from the database at once?

Introduces an upstream dependency
: Creates a dependency on another resource that might fail.

Requires initialization
: What if the `Transactions` table doesn't exist?

Increases complexity
: Yet another thing to manage and configure.

Working with databases in Go

Databases, particularly SQL and SQL-like databases, are everywhere. You can try to avoid them, but if you're building applications with some kind of data component, you will at some point have to interact with one.

Fortunately for us, the creators of the Go standard library provided the `database/sql` package, which provides an idiomatic and lightweight interface around SQL (and SQL-like) databases. In this section we'll briefly demonstrate how to use this package, and point out some of the gotchas along the way.

Among the most ubiquitous members of the `database/sql` package is `sql.DB`: Go's primary database abstraction and entry point for creating statements and transactions, executing queries, and fetching results. While it doesn't, as its name might suggest, map to any particular concept of a database or schema, it does do quite a lot of things for you, including, but not limited to, negotiating connections with your database and managing a database connection pool.

We'll get into how you create your `sql.DB` in a bit. But first, we have to talk about database drivers.

Importing a database driver

While the `sql.DB` type provides a common interface for interacting with a SQL database, it depends on database drivers to implement the specifics for particular database types. At the time of this writing there are 45 drivers listed in the Go repository.

In the following section we'll be working with a Postgres database, so we'll use the third-party `lib/pq` Postgres driver implementation.

To load a database driver, anonymously import the driver package by aliasing its package qualifier to _. This triggers any initializers the package might have while also informing the compiler that you have no intention of directly using it:

```
import (
    "database/sql"
    _ "github.com/lib/pq"       // Anonymously import the driver package
)
```

Now that you've done this, you're finally ready to create your `sql.DB` value and access the database.

Implementing your PostgresTransactionLogger

Previously, we presented the `TransactionLogger` interface, which provides a standard definition for a generic transaction log. You might recall that it defined methods for starting the logger, as well as reading and writing events to the log, as detailed here:

```
type TransactionLogger interface {
    WriteDelete(key string)
    WritePut(key, value string)
    Err() <-chan error

    ReadEvents() (<-chan Event, <-chan error)

    Run()
}
```

Our goal now is to create a database-backed implementation of `TransactionLogger`. Fortunately, much of our work is already done for us. Looking back at "Implementing your FileTransactionLogger" on page 129 for guidance, it looks like we can create a `PostgresTransactionLogger` using very similar logic.

Starting with the `WritePut`, `WriteDelete`, and `Err` methods, you can do something like the following:

```
type PostgresTransactionLogger struct {
    events       chan<- Event       // Write-only channel for sending events
    errors       <-chan error       // Read-only channel for receiving errors
    db           *sql.DB            // The database access interface
}

func (l *PostgresTransactionLogger) WritePut(key, value string) {
    l.events <- Event{EventType: EventPut, Key: key, Value: value}
}

func (l *PostgresTransactionLogger) WriteDelete(key string) {
    l.events <- Event{EventType: EventDelete, Key: key}
}

func (l *PostgresTransactionLogger) Err() <-chan error {
    return l.errors
}
```

If you compare this to the `FileTransactionLogger` it's clear that the code is nearly identical. All we've really changed is:

- Renaming (obviously) the type to `PostgresTransactionLogger`
- Swapping the `*os.File` for a `*sql.DB`
- Removing `lastSequence`; you can let the database handle the sequencing

Creating a new PostgresTransactionLogger

That's all well and good, but we still haven't talked about how we create the `sql.DB`. I know how you must feel. The suspense is definitely killing me, too.

Much like we did in the `NewFileTransactionLogger` function, we're going to create a construction function for our `PostgresTransactionLogger`, which we'll call (quite predictably) `NewPostgresTransactionLogger`. However, instead of opening a file like `NewFileTransactionLogger`, it'll establish a connection with the database, returning an `error` if it fails.

There's a little bit of a wrinkle, though. Namely, that the setup for a Postgres connection takes a lot of parameters. At the bare minimum we need to know the host where the database lives, the name of the database, and the user name and password. One way to deal with this would be to create a function like the following, which simply accepts a bunch of string parameters:

```
func NewPostgresTransactionLogger(host, dbName, user, password string)
    (TransactionLogger, error) { ... }
```

This approach is pretty ugly, though. Plus, what if you wanted an additional parameter? Do you chunk it onto the end of the parameter list, breaking any code that's

already using this function? Maybe worse, the parameter order isn't clear without looking at the documentation.

There has to be a better way. So, instead of this potential horror show, you can create a small helper struct:

```
type PostgresDBParams struct {
    dbName   string
    host     string
    user     string
    password string
}
```

Unlike the big-bag-of-strings approach, this struct is small, readable, and easily extended. To use it, you can create a `PostgresDBParams` variable and pass it to your construction function. Here's what that looks like:

```
logger, err = NewPostgresTransactionLogger(PostgresDBParams{
    host:     "localhost",
    dbName:   "kvs",
    user:     "test",
    password: "hunter2"
})
```

The new construction function looks something like the following:

```
func NewPostgresTransactionLogger(config PostgresDBParams) (TransactionLogger,
    error) {

    connStr := fmt.Sprintf("host=%s dbname=%s user=%s password=%s",
        config.host, config.dbName, config.user, config.password)

    db, err := sql.Open("postgres", connStr)
    if err != nil {
        return nil, fmt.Errorf("failed to open db: %w", err)
    }

    err = db.Ping()                 // Test the database connection
    if err != nil {
        return nil, fmt.Errorf("failed to open db connection: %w", err)
    }

    logger := &PostgresTransactionLogger{db: db}

    exists, err := logger.verifyTableExists()
    if err != nil {
        return nil, fmt.Errorf("failed to verify table exists: %w", err)
    }
    if !exists {
        if err = logger.createTable(); err != nil {
            return nil, fmt.Errorf("failed to create table: %w", err)
        }
    }
```

```
    return logger, nil
}
```

This does quite a few things, but fundamentally it is not very different from `NewFileTransactionLogger`.

The first thing it does is to use `sql.Open` to retrieve a `*sql.DB` value. You'll note that the connection string passed to `sql.Open` contains several parameters; the `lib/pq` package supports many more than the ones listed here. See the package documentation for a complete listing.

Many drivers, including `lib/pq`, don't actually create a connection to the database immediately, so it uses `db.Ping` to force the driver to establish and test a connection.

Finally, it creates the `PostgresTransactionLogger` and uses that to verify that the `transactions` table exists, creating it if necessary. Without this step, the `PostgresTransactionLogger` will essentially assume that the table already exists, and will fail if it doesn't.

You may have noticed that the `verifyTableExists` and `createTable` methods aren't implemented here. This is entirely intentional. As an exercise, you're encouraged to dive into the `database/sql` docs and think about how you might go about doing that. If you'd prefer not to, you can find an implementation in the GitHub repository that comes with this book.

You now have a construction function that establishes a connection to the database and returns a newly created `TransactionLogger`. But, once again, you need to get things started. For that, you need to implement the `Run` method that will create the `events` and `errors` channels and spawn the event ingestion goroutine.

Using db.Exec to execute a SQL INSERT

For the `FileTransactionLogger`, you implemented a `Run` method that initialized the channels and created the go function responsible for writing to the transaction log.

The `PostgresTransactionLogger` is very similar. However, instead of appending a line to a file, the new logger uses `db.Exec` to execute an SQL `INSERT` to accomplish the same result:

```
func (l *PostgresTransactionLogger) Run() {
    events := make(chan Event, 16)              // Make an events channel
    l.events = events

    errors := make(chan error, 1)               // Make an errors channel
    l.errors = errors

    go func() {                                 // The INSERT query
        query := `INSERT INTO transactions
```

```
            (event_type, key, value)
            VALUES ($1, $2, $3)`

        for e := range events {                 // Retrieve the next Event

            _, err := l.db.Exec(                // Execute the INSERT query
                query,
                e.EventType, e.Key, e.Value)

            if err != nil {
                errors <- err
            }
        }
    }()
}
```

This implementation of the `Run` method does almost exactly what its `FileTransactionLogger` equivalent does: it creates the buffered `events` and `errors` channels, and it starts a goroutine that retrieves `Event` values from our `events` channel and writes them to the transaction log.

Unlike the `FileTransactionLogger`, which appends to a file, this goroutine uses `db.Exec` to execute a SQL query that appends a row to the `transactions` table. The numbered arguments (`$1, $2, $3`) in the query are placeholder query parameters, which must be satisfied when the `db.Exec` function is called.

Using db.Query to play back postgres transaction logs

In "Using a bufio.Scanner to play back file transaction logs" on page 133, you used a `bufio.Scanner` to read previously written transaction log entries.

The Postgres implementation won't be *quite* as straightforward, but the principle is the same: you point at the top of your data source and read until you hit the bottom:

```
func (l *PostgresTransactionLogger) ReadEvents() (<-chan Event, <-chan error) {
    outEvent := make(chan Event)               // An unbuffered events channel
    outError := make(chan error, 1)            // A buffered errors channel

    go func() {
        defer close(outEvent)                  // Close the channels when the
        defer close(outError)                  // goroutine ends

        query := `SELECT sequence, event_type, key, value FROM transactions
                  ORDER BY sequence`

        rows, err := db.Query(query)           // Run query; get result set
        if err != nil {
            outError <- fmt.Errorf("sql query error: %w", err)
            return
        }
```

```
        defer rows.Close()                    // This is important!

        e := Event{}                          // Create an empty Event

        for rows.Next() {                     // Iterate over the rows

            err = rows.Scan(                  // Read the values from the
                &e.Sequence, &e.EventType,    // row into the Event.
                &e.Key, &e.Value)

            if err != nil {
                outError <- fmt.Errorf("error reading row: %w", err)
                return
            }

            outEvent <- e                     // Send e to the channel
        }

        err = rows.Err()
        if err != nil {
            outError <- fmt.Errorf("transaction log read failure: %w", err)
        }
    }()

    return outEvent, outError
}
```

All of the interesting (or at least new) bits are happening in the goroutine. Let's break them down:

- `query` is a string that contains the SQL query. The query in this code requests four columns: `sequence`, `event_type`, `key`, and `value`.
- `db.Query` sends `query` to the database, and returns values of type `*sql.Rows` and `error`.
- We defer a call to `rows.Close`. Failing to do so can lead to connection leakage!
- `rows.Next` lets us iterate over the rows; it returns `false` if there are no more rows or if there's an error.
- `rows.Scan` copies the columns in the current row into the values we pointed at in the call.
- We send event `e` to the output channel.
- `Err` returns the error, if any, that may have caused `rows.Next` to return `false`.

Initializing the PostgresTransactionLogger in your web service

The `PostgresTransactionLogger` is pretty much complete. Now let's go ahead and integrate it into the web service.

Fortunately, since we already had the `FileTransactionLogger` in place, we only need to change one line:

```
logger, err = NewFileTransactionLogger("transaction.log")
```

which becomes…

```
logger, err = NewPostgresTransactionLogger("localhost")
```

Yup. That's it. Really.

Because this represents a complete implementation of the `TransactionLogger` interface, everything else stays exactly the same. You can interact with the `PostgresTransactionLogger` using exactly the same methods as before.

Future improvements

As with the `FileTransactionLogger`, the `PostgresTransactionLogger` represents a minimal viable implementation of a transaction logger and has lots of room for improvement. Some of the areas for improvement include, but are certainly not limited to:

- We assume that the database and table exist, and we'll get errors if they don't.
- The connection string is hard-coded. Even the password.
- There's still no `Close` method to clean up open connections.
- The service can close with events still in the write buffer: events can get lost.
- The log retains records of deleted values forever: it will grow indefinitely.

All of these would be (major) impediments in production. I encourage you to take the time to consider—or even implement—solutions to one or more of these points.

Generation 3: Implementing Transport Layer Security

Security. Love it or hate it, the simple fact is that security is a critical feature of *any* application, cloud native or otherwise. Sadly, security is often treated as an afterthought, with potentially catastrophic consequences.

There are rich tools and established security best practices for traditional environments, but this is less true of cloud native applications, which tend to take the form of several small, often ephemeral, microservices. While this architecture provides significant flexibility and scalability benefits, it also creates a distinct opportunity for would-be attackers: every communication between services is transmitted across a network, opening it up to eavesdropping and manipulation.

The subject of security can take up an entire book of its own,[14] so we'll focus on one common technique: encryption. Encrypting data "in transit" (or "on the wire") is commonly used to guard against eavesdropping and message manipulation, and any language worth its salt—including, and especially, Go—will make it relatively low-lift to implement.

Transport Layer Security

Transport Layer Security (TLS) is a cryptographic protocol that's designed to provide communications security over a computer network. Its use is ubiquitous and widespread, being applicable to virtually any Internet communications. You're most likely familiar with it (and perhaps using it right now) in the form of HTTPS—also known as HTTP over TLS—which uses TLS to encrypt exchanges over HTTP.

TLS encrypts messages using *public-key cryptography*, in which both parties possess their own *key pair*, which includes a *public key* that's freely given out, and a *private key* that's known only to its owner, illustrated in Figure 5-2. Anybody can use a public key to encrypt a message, but it can only be decrypted with the corresponding private key. Using this protocol, two parties that wish to communicate privately can exchange their public keys, which can then be used to secure all subsequent communications in a way that can only be read by the owner of the intended recipient, who holds the corresponding private key.[15]

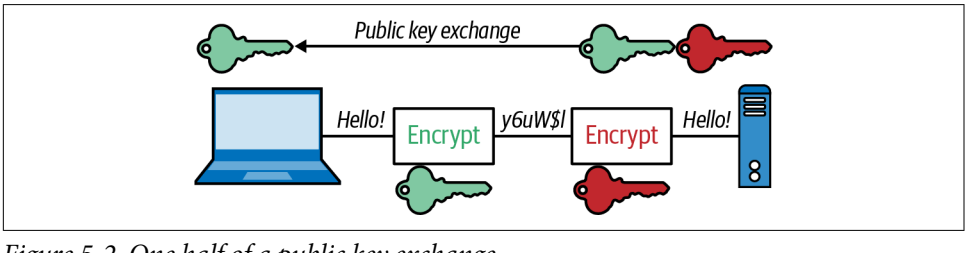

Figure 5-2. One half of a public key exchange

Certificates, certificate authorities, and trust

If TLS had a motto, it would be "trust but verify." Actually, scratch the trust part. Verify everything.

It's not enough for a service to simply provide a public key.[16] Instead, every public key is associated with a *digital certificate*, an electronic document used to prove the key's

14 Ideally written by somebody who knows more than I do about security.

15 This is a gross over-simplification, but it'll do for our purposes. I encourage you to learn more about this and correct me, though.

16 You don't know where that key has been.

ownership. A certificate shows that the owner of the public key is, in fact, the named subject (owner) of the certificate, and describes how the key may be used. This allows the recipient to compare the certificate against various "trusts" to decide whether it will accept it as valid.

First, the certificate must be digitally signed and authenticated by a *certificate authority*, a trusted entity that issues digital certificates.

Second, the subject of the certificate has to match the domain name of the service the client is trying to connect to. Among other things, this helps to ensure that the certificates you're receiving are valid and haven't been swapped out by a man-in-the-middle.

Only then will your conversation proceed.

Web browsers or other tools will usually allow you to choose to proceed if a certificate can't be validated. If you're using self-signed certificates for development, for example, that might make sense. But generally speaking, heed the warnings.

Private Key and Certificate Files

TLS (and its predecessor, SSL) has been around long enough[17] that you'd think that we'd have settled on a single key container format, but you'd be wrong. Web searches for "key file format" will return a virtual zoo of file extensions: *.csr*, *.key*, *.pkcs12*, *.der*, and *.pem* just to name a few.

Of these, however, *.pem* seems to be the most common. It also happens to be the format that's most easily supported by Go's `net/http` package, so that's what we'll be using.

Privacy enhanced mail (PEM) file format

Privacy enhanced mail (PEM) is a common certificate container format, usually stored in *.pem* files, but *.cer* or *.crt* (for certificates) and *.key* (for public or private keys) are common too. Conveniently, PEM is also base64 encoded and therefore viewable in a text editor, and even safe to paste into (for example) the body of an email message.[18]

17 SSL 2.0 was released in 1995 and TLS 1.0 was released in 1999. Interestingly, SSL 1.0 had some pretty profound security flaws and was never publicly released.

18 Public keys only, please.

Often, *.pem* files will come in a pair, representing a complete key pair:

cert.pem
: The server certificate (including the CA-signed public key).

key.pem
: A private key, not to be shared.

Going forward, we'll assume that your keys are in this configuration. If you don't yet have any keys and need to generate some for development purposes, instructions are available in multiple places online. If you already have a key file in some other format, converting it is beyond the scope of this book. However, the Internet is a magical place, and there are plenty of tutorials online for converting between common key formats.

Securing Your Web Service with HTTPS

So, now that we've established that security should be taken seriously, and that communication via TLS is a bare-minimum first step towards securing our communications, how do we go about doing that?

One way might be to put a reverse proxy in front of our service that can handle HTTPS requests and forward them to our key-value service as HTTP, but unless the two are co-located on the same server, we're still sending unencrypted messages over a network. Plus, the additional service adds some architectural complexity that we might prefer to avoid. Perhaps we can have our key-value service serve HTTPS?

Actually, we can. Going all the way back to "Building an HTTP Server with net/http" on page 112, you might recall that the `net/http` package contains a function, `ListenAndServe`, which, in its most basic form, looks something like the following:

```
func main() {
    http.HandleFunc("/", helloGoHandler)            // Add a root path handler

    http.ListenAndServe(":8080", nil)               // Start the HTTP server
}
```

In this example, we call `HandleFunc` to add a handler function for the root path, followed by `ListenAndServe` to start the service listening and serving. For the sake of simplicity, we ignore any errors returned by `ListenAndServe`.

There aren't a lot of moving parts here, which is kind of nice. In keeping with that philosophy, the designers of `net/http` kindly provided a TLS-enabled variant of the `ListenAndServe` function that we're familiar with:

```
func ListenAndServeTLS(addr, certFile, keyFile string, handler Handler) error
```

As you can see, `ListenAndServeTLS` looks and feels almost exactly like `ListenAndServe` except that it has two extra parameters: `certFile` and `keyFile`. If you happen to have certificate and private key PEM files, then service HTTPS-encrypted connections is just a matter of passing the names of those files to `ListenAndServeTLS`:

```
http.ListenAndServeTLS(":8080", "cert.pem", "key.pem", nil)
```

This sure looks super convenient, but does it work? Let's fire up our service (using self-signed certificates) and find out.

Dusting off our old friend `curl`, let's try inserting a key/value pair. Note that we use the `https` scheme in our URL instead of `http`:

```
$ curl -X PUT -d 'Hello, key-value store!' -v https://localhost:8080/v1/key-a
* SSL certificate problem: self signed certificate
curl: (60) SSL certificate problem: self signed certificate
```

Well, that didn't go as planned. As we mentioned in "Certificates, certificate authorities, and trust" on page 146, TLS expects any certificates to be signed by a certificate authority. It doesn't like self-signed certificates.

Fortunately, we can turn this safety check off in `curl` with the appropriately named `--insecure` flag:

```
$ curl -X PUT -d 'Hello, key-value store!' --insecure -v \
    https://localhost:8080/v1/key-a
* SSL certificate verify result: self signed certificate (18), continuing anyway.
> PUT /v1/key-a HTTP/2
< HTTP/2 201
```

We got a sternly worded warning, but it worked!

Transport Layer Summary

We've covered quite a lot in just a few pages. The topic of security is vast, and there's no way we're going to do it justice, but we were able to at least introduce TLS, and how it can serve as one relatively low-cost, high-return component of a larger security strategy.

We were also able to demonstrate how to implement TLS in an Go `net/http` web service, and saw how—as long as we have valid certificates—to secure a service's communications without a great deal of effort.

Containerizing Your Key-Value Store

A *container* is a lightweight operating-system-level virtualization[19] abstraction that provides processes with a degree of isolation, both from their host and from other containers. The concept of the container has been around since at least 2000, but it was the introduction of Docker in 2013 that made containers accessible to the masses and brought containerization into the mainstream.

Importantly, containers are not virtual machines:[20] they don't use hypervisors, and they share the host's kernel rather than carrying their own guest operating system. Instead, their isolation is provided by a clever application of several Linux kernel features, including `chroot`, cgroups, and kernel namespaces. In fact, it can be reasonably argued that containers are nothing more than a convenient abstraction, and that there's actually no such thing as a container.

Even though they're not virtual machines,[21] containers do provide some virtual-machine-like benefits. The most obvious of which is that they allow an application, its dependencies, and much of its environment to be packaged within a single distributable artifact—a container image—that can be executed on any suitable host.

The benefits don't stop there, however. In case you need them, here's a few more:

Agility
: Unlike virtual machines that are saddled with an entire operating system and a colossal memory footprint, containers boast image sizes in the megabyte range and startup times that measure in milliseconds. This is particularly true of Go applications, whose binaries have few, if any, dependencies.

Isolation
: This was hinted at previously, but bears repeating. Containers virtualize CPU, memory, storage, and network resources at the operating-system-level, providing developers with a sandboxed view of the OS that is logically isolated from other applications.

Standardization and productivity
: Containers let you package an application alongside its dependencies, such as specific versions of language runtimes and libraries, as a single distributable binary, making your deployments reproducible, predictable, and versionable.

19 Containers are not virtual machines. They virtualize the operating system instead of hardware.

20 Repetition intended. This is an important point.

21 Yup. I said it. Again.

Orchestration
: Sophisticated container orchestration systems like Kubernetes provide a huge number of benefits. By containerizing your application(s) you're taking the first step towards being able to take advantage of them.

There are just four (very) motivating arguments.[22] In other words, containerization is super, super useful.

For this book, we'll be using Docker to build our container images. Alternative build tools exist, but Docker is the most common containerization tool in use today, and the syntax for its build file—termed a *Dockerfile*—lets you use familiar shell scripting commands and utilities.

That being said, this isn't a book about Docker or containerization, so our discussion will mostly be limited to the bare basics of using Docker with Go. If you're interested in learning more, I suggest picking up a copy of *Docker: Up & Running: Shipping Reliable Containers in Production* by Sean P. Kane and Karl Matthias (O'Reilly).

Docker (Absolute) Basics

Before we continue, it's important to draw a distinction between container images and the containers themselves. A *container image* is essentially an executable binary that contains your application runtime and its dependencies. When an image is run, the resulting process is the *container*. An image can be run many times to create multiple (essentially) identical containers.

Over the next few pages we'll create a simple Dockerfile and build and execute an image. If you haven't already, please take a moment and install the Docker Community Edition (CE).

The Dockerfile

Dockerfiles are essentially build files that describe the steps required to build an image. A very minimal—but complete—example is demonstrated in the following:

```
# The parent image. At build time, this image will be pulled and
# subsequent instructions run against it.
FROM ubuntu:20.04

# Update apt cache and install nginx without an approval prompt.
RUN apt-get update && apt-get install --yes nginx

# Tell Docker this image's containers will use port 80.
EXPOSE 80
```

22 The initial draft had several more, but this chapter is already pretty lengthy.

```
# Run Nginx in the foreground. This is important: without a
# foreground process the container will automatically stop.
CMD ["nginx", "-g", "daemon off;"]
```

As you can see, this Dockerfile includes four different commands:

`FROM`
: Specifies a *base image* that this build will extend, and will typically be a common Linux distribution, such as `ubuntu` or `alpine`. At build time this image is pulled and run, and the subsequent commands applied to it.

`RUN`
: Will execute any commands on top of the current image. The result will be used for the next step in the Dockerfile.

`EXPOSE`
: Tells Docker which port(s) the container will use. See "What's the Difference Between Exposing and Publishing Ports?" on page 154 for more information on exposing ports.

`CMD`
: The command to execute when the container is executed. There can only be one `CMD` in a Dockerfile.

These are four of the most common Dockerfile instructions of many available. For a complete listing see the official Dockerfile reference.

As you may have inferred, the previous example starts with an existing Linux distribution image (Ubuntu 20.04) and installs Nginx, which is executed when the container is started.

By convention, the file name of a Dockerfile is *Dockerfile*. Go ahead and create a new file named *Dockerfile* and paste the previous example into it.

Building your container image

Now that you have a simple Dockerfile, you can build it! Make sure that you're in the same directory as your Dockerfile and enter the following:

```
$ docker build --tag my-nginx .
```

This will instruct Docker to begin the build process. If everything works correctly (and why wouldn't it?) you'll see the output as Docker downloads the parent image, and runs the `apt` commands. This will probably take a minute or two the first time you run it.

At the end, you'll see a line that looks something like the following: `Successfully tagged my-nginx:latest`.

If you do, you can use the `docker images` command to verify that your image is now present. You should see something like the following:

```
$ docker images
REPOSITORY      TAG         IMAGE ID          CREATED              SIZE
my-nginx        latest      64ea3e21a388      29 seconds ago       159MB
ubuntu          20.04       f63181f19b2f      3 weeks ago          72.9MB
```

If all has gone as planned, you'll see at least two images listed: our parent image `ubuntu:20.04`, and your own `my-nginx:latest` image. Next step: running the service container!

What Does latest Mean?

Note the name of the image. What's `latest`? That's a simple question with a complicated answer. Docker images have two name components: a `repository` and a `tag`.

The repository name component can include the domain name of a host where the image is stored (or will be stored). For example, the repository name for an image hosted by FooCorp may be named something like `docker.foo.com/ubuntu`. If no repository URL is evident, then the image is either 100% local (like the image we just built) or lives in the Docker Hub.

The `tag` component is intended as a unique label for a particular version of an image, and often takes the form of a version number. The `latest` tag is a default tag name that's added by many `docker` operations if no tag is specified.

Using `latest` in production is generally considered a bad practice, however, because its contents can change—sometimes significantly—with unfortunate consequences.

Running your container image

Now that you've built your image, you can run it. For that, you'll use the `docker run` command:

```
$ docker run --detach --publish 8080:80 --name nginx my-nginx
61bb4d01017236f6261ede5749b421e4f65d43cb67e8e7aa8439dc0f06afe0f3
```

This instructs Docker to run a container using your `my-nginx` image. The `--detach` flag will cause the container to be run in the background. Using `--publish 8080:80` instructs Docker to publish port 8080 on the host bridged to port 80 in the container, so any connections to `localhost:8080` will be forwarded to the container's port 80. Finally, the `--name nginx` flag specifies a name for the container; without this, a randomly generated name will be assigned instead.

You'll notice that running this command presents us with a very cryptic line containing 65 very cryptic hexadecimal characters. This is the *container ID*, which can be used to refer to the container in lieu of its name.

What's the Difference Between Exposing and Publishing Ports?

The difference between "exposing" and "publishing" container ports can be confusing, but there's actually an important distinction:

- *Exposing* ports is a way of clearly documenting—both to users and to Docker—which ports a container uses. It does not map or open any ports on the host. Ports can be exposed using the `EXPOSE` keyword in the Dockerfile or the `--expose` flag to `docker run`.
- *Publishing* ports tells Docker which ports to open on the container's network interface. Ports can be published using the `--publish` or `--publish-all` flag to `docker run`, which create firewall rules that map a container port to a port on the host.

Running your container image

To verify that your container is running and is doing what you expect, you can use the `docker ps` command to list all running containers. This should look something like the following:

```
$ docker ps
CONTAINER ID    IMAGE       STATUS          PORTS                   NAMES
4cce9201f484    my-nginx    Up 4 minutes    0.0.0.0:8080->80/tcp    nginx
```

The preceding output has been edited for brevity (you may notice that it's missing the `COMMAND` and `CREATED` columns). Your output should include seven columns:

`CONTAINER ID`
: The first 12 characters of the container ID. You'll notice it matches the output of your `docker run`.

`IMAGE`
: The name (and tag, if specified) of this container's source image. No tag implies `latest`.

`COMMAND` *(not shown)*
: The command running inside the container. Unless overridden in the `docker run` this will be the same as the `CMD` instruction in the Dockerfile. In our case this will be `nginx -g 'daemon off;'`.

`CREATED` *(not shown)*
: How long ago the container was created.

`STATUS`
: The current state of the container (`up`, `exited`, `restarting`, etc) and how long it's been in that state. If the state changed, then the time will differ from `CREATED`.

`PORTS`
: Lists all exposed and published ports (see "What's the Difference Between Exposing and Publishing Ports?" on page 154). In our case, we've published `0.0.0.0:8080` on the host and mapped it to `80` on the container, so that all requests to host port 8080 are forwarded to container port 80.

`NAMES`
: The name of the container. Docker will randomly set this if it's not explicitly defined. Two containers with the same name, regardless of state, cannot exist on the same host at the same time. To reuse a name, you'll first have to `delete` the unwanted container.

Issuing a request to a published container port

If you've gotten this far, then your `docker ps` output should show a container named `nginx` that appears to have port 8080 published and forwarding to the container's port 80. If so, then you're ready to send a request to your running container. But which port should you query?

Well, the Nginx container is listening on port 80. Can you reach that? Actually, no. That port won't be accessible because it wasn't published to any network interface during the `docker run`. Any attempt to connect to an unpublished container port is doomed to failure:

```
$ curl localhost:80
curl: (7) Failed to connect to localhost port 80: Connection refused
```

You haven't published to port 80, but you *have* published port 8080 and forwarded it to the container's port 80. You can verify this with our old friend `curl` or by browsing to `localhost:8080`. If everything is working correctly you'll be greeted with the familiar Nginx "Welcome" page illustrated in Figure 5-3.

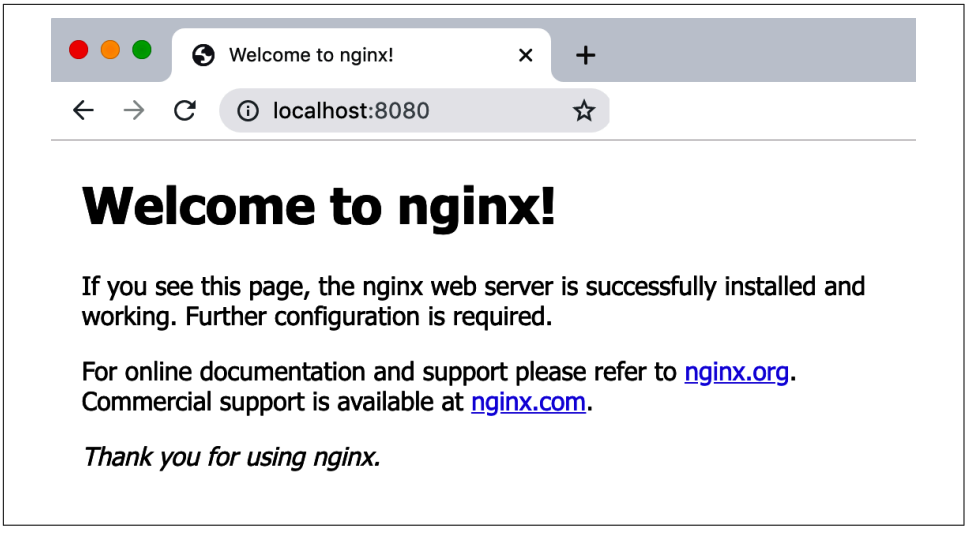

Figure 5-3. Welcome to nginx!

Running multiple containers

One of the "killer features" of containerization is this: because all of the containers on a host are isolated from one another, it's possible to run quite a lot of them—even ones that contain different technologies and stacks—on the same host, with each listening on a different published port. For example, if you wanted to run an `httpd` container alongside your already-running `my-nginx` container, you could do exactly that.

"But," you might say, "both of those containers expose port 80! Won't they collide?"

Great question, to which the answer is, happily, no. In fact, you can actually have as many containers as you want that *expose* the same port—even multiple instances of the same image—as long as they don't attempt to *publish* the same port on the same network interface.

For example, if you want to run the stock `httpd` image, you can run it by using the `docker run` command again, as long as you take care to publish to a different port (8081, in this case):

```
$ docker run --detach --publish 8081:80 --name httpd httpd
```

If all goes as planned, this will spawn a new container listening on the host at port 8081. Go ahead: use `docker ps` and `curl` to test:

```
$ curl localhost:8081
<html><body><h1>It works!</h1></body></html>
```

Stopping and deleting your containers

Now you've successfully run your container, you'll probably need to stop and delete it at some point, particularly if you want to rerun a new container using the same name.

To stop a running container, you can use the `docker stop` command, passing it either the container name or the first few characters of its container ID (how many characters doesn't matter, as long they can be used to uniquely identify the desired container). Using the container ID to stop our `nginx` container looks like this:

```
$ docker stop 4cce      # "docker stop nginx" will work too
4cce
```

The output of a successful `docker stop` is just the name or ID that we passed into the command. You can verify that your container has actually been stopped using `docker ps --all`, which will show *all* containers, not just the running ones:

```
$ docker ps
CONTAINER ID    IMAGE       STATUS                      PORTS    NAMES
4cce9201f484    my-nginx    Exited (0) 3 minutes ago             nginx
```

If you ran the `httpd` container, it will also be displayed with a status of `Up`. You will probably want to stop it as well.

As you can see, the status of our `nginx` container has changed to `Exited`, followed by its exit code—an exit status of 0 indicates that we were able to execute a graceful shutdown—and how long ago the container entered its current status.

Now that you've stopped your container you can freely delete it.

You can't delete a running container or an image that's used by a running container.

To do this, you use the `docker rm` (or the newer `docker container rm`) command to remove your container, again passing it either the container name or the first few characters of the ID of the container you want to delete:

```
$ docker rm 4cce          # "docker rm nginx" will work too
4cce
```

As before, the output name or ID indicates success. If you were to go ahead and run `docker ps --all` again, you shouldn't see the container listed anymore.

Building Your Key-Value Store Container

Now that you have the basics down, you can start applying them to containerizing our key-value service.

Fortunately, Go's ability to compile into statically linked binaries makes it especially well suited for containerization. While most other languages have to be built into a parent image that contains the language runtime, like the 486MB `openjdk:15` for Java or the 885MB `python:3.9` for Python,[23] Go binaries need no runtime at all. They can be placed into a "scratch" image: an image, with no parent at all.

Iteration 1: adding your binary to a FROM scratch image

To do this, you'll need a Dockerfile. The following example is a pretty typical example of a Dockerfile for a containerized Go binary:

```
# We use a "scratch" image, which contains no distribution files. The
# resulting image and containers will have only the service binary.
FROM scratch

# Copy the existing binary from the host.
COPY kvs .

# Copy in your PEM files.
COPY *.pem .

# Tell Docker we'll be using port 8080.
EXPOSE 8080

# Tell Docker to execute this command on a `docker run`.
CMD ["/kvs"]
```

This Dockerfile is fairly similar to the previous one, except that instead of using `apt` to install an application from a repository, it uses `COPY` to retrieve a compiled binary from the filesystem it's being built on. In this case, it assumes the presence of a binary named `kvs`. For this to work, we'll need to build the binary first.

In order for your binary to be usable inside a container, it has to meet a few criteria:

- It has to be compiled (or cross-compiled) for Linux.
- It has to be statically linked.
- It has to be named `kvs` (because that's what the Dockerfile is expecting).

We can do all of these things in one command, as follows:

23 To be fair, these images are "only" 240MB and 337MB compressed, respectively.

```
$ CGO_ENABLED=0 GOOS=linux go build -a -o kvs
```

Let's walk through what this does:

- `CGO_ENABLED=0` tells the compiler to disable `cgo` and statically link any C bindings. We won't go into what this is, other than that it enforces static linking, but I encourage you to look at the cgo documentation if you're curious.
- `GOOS=linux` instructs the compiler to generate a Linux binary, cross-compiling if necessary.
- `-a` forces the compiler to rebuild any packages that are already up to date.
- `-o kvs` specifies that the binary will be named `kvs`.

Executing the command should yield a statically linked Linux binary. This can be verified using the `file` command:

```
$ file kvs
kvs: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked,
not stripped
```

Linux binaries will run in a Linux container, even one running in Docker for MacOS or Windows, but won't run on MacOS or Windows otherwise.

Great! Now let's build the container image, and see what comes out:

```
$ docker build --tag kvs .
...output omitted.

$ docker images
REPOSITORY     TAG        IMAGE ID           CREATED              SIZE
kvs            latest     7b1fb6fa93e3       About a minute ago   6.88MB
node           15         ebcfbb59a4bd       7 days ago           936MB
python         3.9        2a93c239d591       8 days ago           885MB
openjdk        15         7666c92f41b0       2 weeks ago          486MB
```

Less than 7MB! That's roughly two orders of magnitude smaller than the relatively massive images for other languages' runtimes. This can come in quite handy when you're operating at scale and have to pull your image onto a couple hundred nodes a few times a day.

But does it run? Let's find out:

```
$ docker run --detach --publish 8080:8080 kvs
4a05617539125f7f28357d3310759c2ef388f456b07ea0763350a78da661afd3

$ curl -X PUT -d 'Hello, key-value store!' -v http://localhost:8080/v1/key-a
> PUT /v1/key-a HTTP/1.1
```

```
< HTTP/1.1 201 Created

$ curl http://localhost:8080/v1/key-a
Hello, key-value store!
```

Looks like it works!

So now you have a nice, simple Dockerfile that builds an image using a precompiled binary. Unfortunately, that means that you have to make sure that you (or your CI system) rebuilds the binary fresh for each Docker build. That's not *too* terrible, but it does mean that you need to have Go installed on your build workers. Again, not terrible, but we can certainly do better.

Iteration 2: using a multi-stage build

In the last section, you created a simple Dockerfile that would take an existing Linux binary and wrap it in a bare-bones "scratch" image. But what if you could perform the *entire* image build—Go compilation and all—in Docker?

One approach might be to use the `golang` image as our parent image. If you did that, your Dockerfile could compile your Go code and run the resulting binary at deploy time. This could build on hosts that don't have the Go compiler installed, but the resulting image would be saddled with an additional 862MB (the size of the `golang:1.16` image) of entirely unnecessary build machinery.

Another approach might be to use two Dockerfiles: one for building the binary, and another that containerizes the output of the first build. This is a lot closer to where you want to be, but it requires two distinct Dockerfiles that need be sequentially built or managed by a separate script.

A better way became available with the introduction of multistage Docker builds, which allow multiple distinct builds—even with entirely different base images—to be chained together so that artifacts from one stage can be selectively copied into another, leaving behind everything you don't want in the final image. To use this approach, you define a build with two stages: a "build" stage that generates the Go binary, and an "image" stage that uses that binary to produce the final image.

To do this, you use multiple `FROM` statements in our Dockerfile, each defining the start of a new stage. Each stage can be arbitrarily named. For example, you might name your build stage `build`, as follows:

```
FROM golang:1.16 as build
```

Once you have stages with names, you can use the `COPY` instruction in your Dockerfile to copy any artifact *from any previous stage*. Your final stage might have an instruction like the following, which copies the file `/src/kvs` from the `build` stage to the current working directory:

```
COPY --from=build /src/kvs .
```

Putting these things together yields a complete, two-stage Dockerfile:

```
# Stage 1: Compile the binary in a containerized Golang environment
#
FROM golang:1.16 as build

# Copy the source files from the host
COPY . /src

# Set the working directory to the same place we copied the code
WORKDIR /src

# Build the binary!
RUN CGO_ENABLED=0 GOOS=linux go build -o kvs


# Stage 2: Build the Key-Value Store image proper
#
# Use a "scratch" image, which contains no distribution files
FROM scratch

# Copy the binary from the build container
COPY --from=build /src/kvs .

# If you're using TLS, copy the .pem files too
COPY --from=build /src/*.pem .

# Tell Docker we'll be using port 8080
EXPOSE 8080

# Tell Docker to execute this command on a "docker run"
CMD ["/kvs"]
```

Now that you have your complete Dockerfile, you can build it in precisely the same way as before. We'll tag it as `multipart` this time, though, so that you can compare the two images:

```
$ docker build --tag kvs:multipart .
...output omitted.

$ docker images
REPOSITORY     TAG           IMAGE ID          CREATED              SIZE
kvs            latest        7b1fb6fa93e3      2 hours ago          6.88MB
kvs            multipart     b83b9e479ae7      4 minutes ago        6.56MB
```

This is encouraging! You now have a single Dockerfile that can compile your Go code—regardless of whether or not the Go compiler is even installed on the build worker—and that drops the resulting statically linked executable binary into a `FROM scratch` base to produce a very, very small image containing nothing except your key-value store service.

You don't need to stop there, though. If you wanted to, you could add other stages as well, such as a `test` stage that runs any unit tests prior to the build step. We won't go through that exercise now, however, since it's more of the same thing, but I encourage you to try it for yourself.

Externalizing Container Data

Containers are intended to be ephemeral, and any container should be designed and run with the understanding that it can (and will) be destroyed and recreated at any time, taking all of its data with it. To be clear, this is a feature, and is very intentional, but sometimes you might *want* your data to outlive your containers.

For example, the ability to mount externally managed files directly into the filesystem of an otherwise general-purpose container can decouple configurations from images so you don't have to rebuild them whenever just to change their settings. This is a very powerful strategy, and is probably the most common use-case for container data externalization. So common in fact that Kubernetes even provides a resource type—`ConfigMap`—dedicated to it.

Similarly, you might want data generated in a container to exist beyond the lifetime of the container. Storing data on the host can be an excellent strategy for warming caches, for example. It's important to keep in mind, however, one of the realities of cloud native infrastructure: nothing is permanent, not even servers. Don't store anything on the host that you don't mind possibly losing forever.

Fortunately, while "pure" Docker limits you to externalizing data directly onto local disk,[24] container orchestration systems like Kubernetes provides various abstractions that allows data to survive the loss of a host.

Unfortunately, this is supposed to be a book about Go, so we really can't cover Kubernetes in detail here. But if you haven't already, I strongly encourage you to take a long look at the excellent Kubernetes documentation, and equally excellent *Kubernetes: Up and Running* by Brendan Burns, Joe Beda, and Kelsey Hightower (O'Reilly).

24 I'm intentionally ignoring solutions like Amazon's Elastic Block Store, which can help, but have issues of their own.

Summary

This was a long chapter, and we touched on a lot of different topics. Consider how much we've accomplished!

- Starting from first principles, we designed and implemented a simple monolithic key-value store, using `net/http` and `gorilla/mux` to build a RESTful service around functionality provided by a small, independent, and easily testable Go library.
- We leveraged Go's powerful interface capabilities to produce two completely different transaction logger implementations, one based on local files and using `os.File` and the `fmt` and `bufio` packages; the other backed by a Postgres database and using the `database/sql` and `github.com/lib/pq` Postgres driver packages.
- We discussed the importance of security in general, covered some of the basics of TLS as one part of a larger security strategy, and implemented HTTPS in our service.
- Finally, we covered containerization, one of the core cloud native technologies, including how to build images and how to run and manage containers. We even containerized not only our application, but we even containerized its build process.

Going forward, we'll be extending on our key-value service in various ways when we introduce new concepts, so stay tuned. Things are about to get even more interesting.