Setting up the Structure of Application

This lesson provides detailed code for designing the application and its explanation.

WE'LL COVER THE FOLLOWING

- Making it thread-safe
- Using `defer` to simplify the code
- A factory for `URLStore`
- Using our `URLStore`

When our application is running in production, it will receive many requests with short URLs, and several requests to turn a long URL into a short one. But, which data structure will our program stock this data in?

Both URL-types (A) and (B) from the last lesson are strings. Moreover, they relate to each other: given (B) as a key, we need to fetch (A) as its value, i.e., they map to each other. To store our data in memory, we need such a map structure, a `map[string]string`. The key type is written between [ ], and the value type follows, as we learned all about maps in Chapter 6. In any non-trivial program, it is useful to give the special types that you use a name. In Go, we do this with the keyword type, so we define a: `type URLStore map[string]string`. It maps short URLs to long URLs; both are strings.

To make a variable of that type named `urls`, just use: `urls := make(URLStore)`. Suppose we want to store the mapping of http://goto/a to http://google.com/ in `urls`; we can do this with the statement:

```
urls ["a"] = "http://google.com/"
```

We just store the suffix of http://goto/ as a key, and what comes before the key is a fixed URL prefix.

To retrieve its corresponding long URL given "a", we write:

To retrieve its corresponding long URL given "a", we write:

```
url := urls ["a"]
```

Then, the value of `url` is equal to "http://google.com/". Note that with `:=` we don't need to say that `url` is of type string; the compiler deduces the type from the value on the right side.

Making it thread-safe

Our `URLStore` variable is the central memory data store here. Once we get some traffic, there will be many requests of type `Redirect`. These are read operations, read with the given short URL as key, and they return the long URL as value. However, requests of type `Add` are different; they change our URLStore, adding new key-value pairs. When our service gets many update type-requests at once, the following problem could arise: the `add` operation could be interrupted by another request of the same type, and the long URL would perhaps not be written as value. Also, there could be modifications together with reads, possibly resulting in having the wrong values read. Our map does not guarantee that once an update starts, it will terminate completely before a new update begins. In other words: a map is not thread-safe, and goto will serve many requests concurrently. Therefore, we must make our `URLStore` type-safe to access from separate threads.

The simplest and classical way to do this is to add a lock to updating operations. In Go, this is a `Mutex` type from the `sync` package in the standard library, which we now must import into our code. We change the type definition of our `URLStore` to a struct type with two fields: the map and an `RWMutex` from the `sync` package:

```
import "sync"
type URLStore struct {
  urls map[string]string // map from short to long URLs
  mu sync.RWMutex
}
```

An `RWMutex` has two locks: one for readers and one for writers. Many clients can take the read lock simultaneously, but only one client can take the write lock (to the exclusion of all readers), effectively serializing the updates and

making them take place consecutively. We will implement our read request

type `Redirect` in a `Get` function, and our write request type `Add` as a `Set` function. The `Get` function looks like this:

```
func (s *URLStore) Get(key string) string {
  s.mu.RLock()
  url := s.urls[key]
  s.mu.RUnlock()
  return url
}
```

It takes a key (short URL) and returns the corresponding map value as a URL. The function works on a variable `s`, which is a pointer to our `URLStore`. But, before reading the value, we set a read-lock with `s.mu.RLock()` so that no update can interrupt the read. After the read, we unlock so that pending updates can start. What if the key is not present in the map? Then the zero value for the string type (an empty string) will be returned. Notice the `.` notation familiar from OO languages: the method `RLock()` is called on the field `mu` of `s`.

The `Set` function needs both a key and a URL and has to use the write lock `Lock()` to exclude any other updates at the same time. It returns a boolean true or false value to indicate whether the Set was successful or not:

```
func (s *URLStore) Set(key, url string) bool {

  s.mu.Lock()
  _, present := s.urls[key]
  if present {
    s.mu.Unlock()
    return false
  }
  s.urls[key] = url
  s.mu.Unlock()
  return true
}
```

With the form `_, present := s.urls[key]`, we can test to see whether our map already contains the key. If so, `present` becomes *true*. Otherwise, it is *false*. This is the so-called *comma, ok* form, which we frequently encounter in Go code. If the key is already present, `Set` returns a boolean false value, and the

code. If the key is already present, `Set` returns a boolean false value, and the map is not updated because we return from the function (so we don't allow our short URL's to be reused). If the key is not `present`, we add it to the map and return true. The `_` on the left side is a placeholder for the value, and it indicates that we are not going to use it because we assign it to `_`. Note that as soon as possible (after the update), we `Unlock()` our `URLStore`.

Using `defer` to simplify the code

In this case, the code is still simple, and it is easy to remember to do the `Unlock()`. However, in more complex code, this might be forgotten or put in the wrong place, leading to problems that are often difficult to track down. For these kinds of situations, Go has a special keyword `defer`, which allows, in this case, the `Unlock` to be signaled immediately after the `Lock`. However, its effect is that the `Unlock()` will only be done just before returning from the function.

`Get` can be simplified to (we have eliminated the local variable URL):

```
func (s *URLStore) Get(key string) string {
  s.mu.RLock()
  defer s.mu.RUnlock()
  return s.urls[key]
}
```

The logic for `Set` also becomes somewhat clearer (we don't need to think about unlocking anymore):

```
func (s *URLStore) Set(key, url string) bool {
  s.mu.Lock()
  defer s.mu.Unlock()
  _, present := s.urls[key]
  if present {
    return false
  }
  s.urls[key] = url
  return true
}
```

A factory for `URLStore`

The `URLStore` struct contains a map field, which must be initialized with make

before it can be used. Making the value of a struct is done in Go by defining a

function with the prefix New, that returns an initialized value of the type (here, and in most cases, a pointer to it):

```
func NewURLStore() *URLStore {
  return &URLStore{ urls: make(map[string]string) }
}
```

In return, we make a `URLStore` literal with our map initialized. The lock doesn't need to be specifically initialized; this is the standard way in Go of making struct objects. `&` is the address-of operator, to return a pointer because `NewURLStore` returns the pointer `*URLStore`. We call this function to make a `URLStore` variable store:

```
var store = NewURLStore()
```

Using our `URLStore`

To add a new short/long URL pair to our map, all we have to do is call the `Set` method on `s`, and since this is a boolean, we can immediately wrap it in an if-statement:

```
if s.Set("a", "http://google.com") {
  // success
}
```

To retrieve the long URL given a short URL, we call the `Get` method on `s` and put the result in a variable `url`:

```
if url := s.Get("a"); url != "" {
  // redirect to url
} else {
  // key not found
}
```

Here, we make use of the fact that, in Go, an `if` can start with an initializing statement before the condition.

We may also need a method `Count` to give us the number of key-value pairs in the map; this is given by the built-in `len` function:

```
func (s *URLStore) Count() int {
  s.mu.RLock()
  defer s.mu.RUnlock()
  return len(s.urls)
}
```

How will we compute the short URL given the long URL? For this, we use a function `genKey(n int) string {...}`; as an argument, we will pass it the current value of `s.Count()`.

The exact algorithm is of little importance. We can now make a `Put` method that takes a long URL, generates its short key with `genKey`, stores the URL under this (short URL) key with the `Set` method, and returns that key:

```
func (s *URLStore) Put(url string) string {
  for {
    key := genKey(s.Count())
    if s.Set(key, url) {
      return key
    }
  }
 // shouldn't get here
return ""
}
```

The for loop retries the `Set` until it is successful (meaning that we have generated a not yet existing short URL).

Until now, we have defined our data structures and functions. In the next lesson, we will define a web server to deliver the Add and the Redirect services.