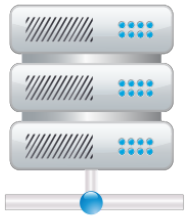
Persistent Storage with gob

This lesson explains how to build persistent storage for the application.



When the `goto` process (the web-server on a port) ends, and this has to happen sooner or later, the map with the shortened URLs in memory will be lost. To preserve our map data, we need to save them to a disk file. We will modify `URLStore` so that it writes its data to a file, and restores that data on `goto` start-up. For this, we will use Go's `encoding/gob` package, which is a serialization and deserialization package that turns data structures into arrays (or more accurately slices) of bytes and vice versa (see Chapter 10).

With the `gob` package's `NewEncoder` and `NewDecoder` functions, you can decide where you write the data to or read it from. The resulting `Encoder` and `Decoder` objects provide `Encode` and `Decode` methods for writing and reading Go data structures to and from files. `Encoder` also implements the `Writer` interface, and so does `Decoder` for the `Reader` interface.

We will add a new file field (of type `*os.File`) to `URLStore` that will be a handle to an open file that can be used for writing and reading.

```
type URLStore struct {
  urls map[string]string
  mu sync.RWMutex
  file *os.File
}
```

We will call this file **store.gob** and give that name as a parameter when we instantiate the `URLStore`:

```
var store = NewURLStore("store.gob")
```

Now, we have to adapt our `NewURLStore` function:

```
func NewURLStore(filename string) *URLStore {
  s := &URLStore{urls: make(map[string]string)}
  f, err := os.OpenFile(filename, os.O_RDWR|os.O_CREATE|os.O_APPEND, 0644)
  if err != nil {
    log.Fatal("URLStore:", err)
  }
  s.file = f
  return s
}
```

The `NewURLStore` function now takes a filename argument, opens the file, and stores the `*os.File` value in the file field of our `URLStore` variable store, here, locally called `s`. The call to `OpenFile` can fail (our disk file could be removed or renamed for example). It can return an error `err`; notice how Go handles this:

```
f, err := os.OpenFile(filename, os.O_RDWR|os.O_CREATE|os.O_APPEND, 0644)
if err != nil {
  log.Fatal("URLStore:", err)
}
```

When `err` is not nil, meaning there was an error, we stop the program with a message. This is one way of doing it: mostly the error is returned to the calling function. However, this pattern of testing errors is ubiquitous in Go code. After the }, we are certain the file is opened. We open this file with writing enabled, more exactly in append-mode. Each time a new (short, long) URL pair is made in our program, we will store it through gob in the file **store.gob**. For that purpose, we define a new struct type `record`:

```
type record struct {
  Key, URL string
}
```

and a new `save` method that writes a given key and URL to disk as a gob-encoded record:

```
func (s *URLStore) save(key, url string) error {
```

```
  e := gob.NewEncoder(s.file)

  return e.Encode(record{key, url})
}
```

At the start of `goto`, our datastore on disk must be read into the `URLStore` map; for this, we have a load method:

```
func (s *URLStore) load() error {
  if _, err := s.file.Seek(0, 0); err != nil {
    return err
  }
  d := gob.NewDecoder(s.file)
  var err error
  for err == nil {
    var r record
    if err = d.Decode(&r); err == nil {
      s.Set(r.Key, r.URL)
    }
  }
  if err == io.EOF {
    return nil
  }
  return err
}
```

The new `load` method will *seek* to the beginning of the file, read and *decode* each record, and store the data in the map using the `Set` method. Again, notice the all-pervasive error-handling. The decoding of the file is an infinite loop which continues as long as there is no error:

```
for err == nil {
  ...
}
```

If we get an error, it could be because we have just decoded the last record and the error `io.EOF` (EndOfFile) occurs. If this is not the case, we have an error while decoding, and this is returned with the return `err`. This method must be added to NewURLStore:

```
func NewURLStore(filename string) *URLStore {
  s := &URLStore{urls: make(map[string]string)}
  f, err := os.OpenFile(filename, os.O_RDWR|os.O_CREATE|os.O_APPEND, 0644)
```

```
  f, err := os.OpenFile(filename, os.O_RDWR|os.O_CREATE|os.O_APPEND, 0644)
  if err != nil {
    log.Fatal("Error opening URLStore:", err)
  }
  s.file = f
  if err := s.load(); err != nil {
    log.Println("Error loading data in URLStore:", err)
  }
  return s
}
```

Also in the `Put` function, when we add a new url-pair to our map, this should also be saved immediately to the datafile:

```
func (s *URLStore) Put(url string) string {
  for {
    key := genKey(s.Count())
    if s.Set(key, url) {
      if err := s.save(key, url); err != nil {
        log.Println("Error saving to URLStore:", err)
      }
      return key
    }
  }
  panic("shouldn't get here")
}
```

Environment Variables

Key:	Value:
GOROOT	/usr/local/go
GOPATH	//root/usr/local/go/src
PATH	//root/usr/local/go/src/bin:/usr/local/go...

```
package main

import (
        "fmt"
        "net/http"
)

var store = NewURLStore("store.gob")

func main() {
        http.HandleFunc("/", Redirect)
        http.HandleFunc("/add", Add)
        http.ListenAndServe(":3000", nil)
}
```

```
func Redirect(w http.ResponseWriter, r *http.Request) {
        key := r.URL.Path[1:]

        url := store.Get(key)
        if url == "" {
                http.NotFound(w, r)
                return
        }
        http.Redirect(w, r, url, http.StatusFound)
}

func Add(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Content-Type", "text/html")
        url := r.FormValue("url")
        if url == "" {
                fmt.Fprint(w, addForm)
                return
        }
        key := store.Put(url)
        fmt.Fprintf(w, "%s", key)
}

const addForm = `
<html><body>
<form method="POST" action="/add">
URL: <input type="text" name="url">
<input type="submit" value="Add">
</form>
</html></body>
`
```

Click **RUN** and type `go run *.go` . Wait for the terminal to start. Once it's started, click the URL next to the **Your app can be found at** and follow the steps taught in the previous lesson.

> **Remark**: To run it locally, change **line 13** to `http.ListenAndServe(":8080", nil)` . To test this program, open a browser at http://localhost:8080.

Goroutines can be used to increase performance. The next lesson explains how to do so.