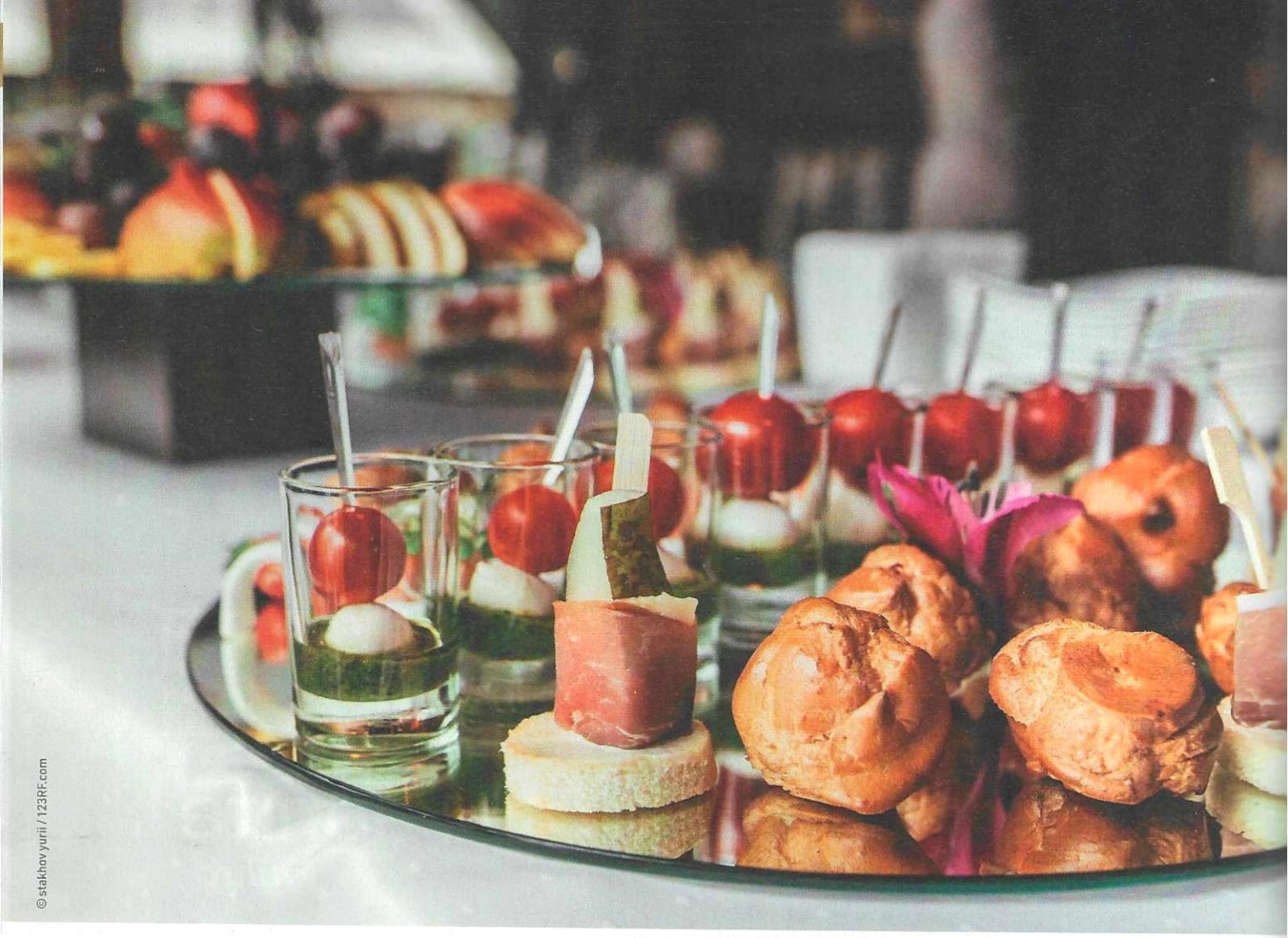


© stakhov yurii / 123RF.com

Go-Programm bereitet Fotos zum Teilen im Web vor

Es ist angerichtet

Um seine begehrten Fotos schnell über einen privaten Link im ansprechenden Layout zum Teilen im Web anzubieten, schreibt Mike Schilli ein Go-Programm, dessen Template-Engine auch die Link-Vorschau im Messenger beherrscht. Michael Schilli

Der Autor

Michael Schilli arbeitet als Software Engineer in der San Francisco Bay Area in Kalifornien. In seiner seit 1997 laufenden Kolumne forscht er jeden Monat nach praktischen Anwendungen verschiedener Programmiersprachen. Unter mschilli@perlmeister.com beantwortet er gern Ihre Fragen.

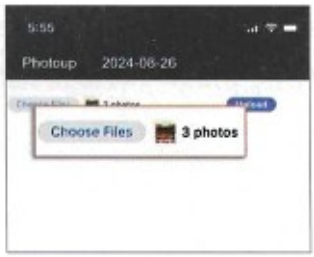
Zeige ich frisch geschossene und gut belichtete Digitalfotos herum, heißt es oft: „Kannst mir das auch gleich schicken?" Als gute Haut, die ich bin, lasse ich mich nicht lange bitten. Handelt es sich um mehrere Fotos, nach denen mehrere Leute lechzen, ist es einfacher, die Kollektion ins Web zu stellen und einen Link darauf zu teilen, auf dass die gierigen Empfänger sich selbst bedienen können.

Diese Snapshot-Folge zeigt ein CGI-Skript `photoup`, das es dem Admin erlaubt, eines oder mehrere Fotos auf einen Webserver hochzuladen. Dort landen sie in einem Verzeichnis mit schwer zu erratendem Namen. Es enthält zudem ein Layout, das die Fotos als Set oder in der Einzelansicht zeigt, und zwar bildschön, ganz egal ob auf dem Desktop oder auf dem Smartphone. Das Ganze ist in Go umgesetzt, mit einer Einführung in Gos Template-Engine, denn das Layout setzt sich aus einzelnen Snippets zusammen, die das fertige Programm als fertige HTML-Seite ausspuckt. Gleichzeitig bindet das Skript mittels Template-Schleifen

auch noch dynamisch Fotos ein. Das reinste Hexenwerk!

Um die während einer Flugreise aufgenommenen Fotos zu teilen, lade ich sie mit dem Browser 1 des Smartphones hoch 2. Der neu generierte Link 3 führt anschließend zu einem Kontaktabzug 4. Ein Mausklick auf ein Bild bringt die JPEG-Datei mit der vollen Auflösung hoch 5, damit Freunde sie nach Herzenslust herunterladen können.

1 Auf dem Mobiltelefon zeigt sich der Uploader formschön und funktional.



2 Drei Fotos stehen hier in der Auswahl zum Hochladen ins Web bereit.

3 Nach dem erfolgreichen Hochladen erscheint ein Link, der geteilt werden kann.

GET oder POST

Das Hochladen der Fotos erledigt das CGI-Programm aus Listing 1. Beim ersten Aufruf zeigt der Browser im Smartphone oder auf dem Desktop das Formular aus Abbildung 1 an. Klickt der Nutzer auf *Choose Files*, fragt der aufpoppende Dialog nach Dateien, entweder aus der Fotosammlung des Smartphones oder einem voreingestellten Ordner auf dem Desktop. Nach der Bestätigung der Auswahl einer oder mehrerer Dateien veranlasst ein Klick auf den Button *Upload* des Formulars den Browser dazu, dieselbe URL anzufordern, was noch einmal das CGI-Programm auf dem Server startet.

Bei jedem Aufruf wirft Listing 1 in Zeile 12 den CGI-Handler ab Zeile 14 an. Wurde dieser wie beim vorher erläuterten Erstkontakt mit der HTTP-Methode GET aufgerufen, malt Zeile 21 das Upload-Formular in den Browser. Drückt der

Listing 1: photoup.go (Fortsetzung auf S. 86)

```
01 package main
02 import (
03   "io"
04   "mime/multipart"
05   "net/http"
06   "net/http/cgi"
07   "os"
08   "path"
09   "regexp"
10 )
11 func main() {
12   cgi.Serve(http.HandlerFunc(uploadHandler))
13 }
14 func uploadHandler(w http.ResponseWriter, r
   *http.Request) {
15   tmpl := NewTmpl()
16   err := tmpl.Init()
17   if err != nil {
18     panic(err)
19   }
20   if r.Method == "GET" {
21     tmpl.RenderPage(w, "upload.html")
22   } else if r.Method == "POST" {
23     link, err := processUpload(w, r)
24     if err != nil {
25       http.Error(w, err.Error(), http.
   StatusInternalServerError)
26       return
27     }
28     tmpl.Link = link
29     tmpl.RenderPage(w, "done.html")
30   }
31 }
32 func processUpload(w http.ResponseWriter, r
   *http.Request) (string, error) {
33   err := r.ParseMultipartForm(10 << 20) // 10 MB
   limit
34   if err != nil {
35     return "", err
36   }
37   files := r.MultipartForm.File["files"]
38   dir, err := uploadDir()
39   if err != nil {
40     return "", err
41   }
42   link := regexp.MustCompile(`/[^/]+/[^/]+$`).
   FindString(dir)
43   names := []string{}
44   for _, fh := range files {
45     name, err := processFile(fh, dir)
46     if err != nil {
47       return "", err
48     }
49     names = append(names, name)
```

4 Ein Kontaktabzug hilft beim Herunterladen im Browser.

5 Ein Klick auf ein Bild bringt die volle Auflösung (oben) und zoomt auf Wunsch hinein (unten).

User aber *Upload*, nutzt der Browser die Methode POST. Zeile 23 ruft daraufhin die Funktion `processUpload()` auf, die ab der Zeile 32 definiert ist. Der Browser hat die ausgewählten Dateien bereits wie gewünscht mitgeschickt, und `ParseMultipartForm()` dekodiert die eingewickelten Dateinamen sowie die dazugehörigen JPEG-Daten der Fotos.

Für jede im Upload gefundene Datei ruft Zeile 45 die Funktion `processFile()` ab Zeile 66 auf. Nach der Reinigung des vorgegebenen Namens mit `sanitizeFileName()` (schließlich darf der Server dem Client nicht trauen) speichert das CGI-Programm die Daten unter dem vorher mit `uploadDir()` erzeugten Verzeichnis. Dieser Pfad ist öffentlich, aber schwer zu erraten (dazu später mehr), also können Bekannte die Fotos von dort als statische Dateien des Webservers herunterladen.

Listing 1: photoup.go (Fortsetzung von S. 85)

```
50   }
51   idx, err := os.OpenFile(path.Join(dir, "index.
     html"), os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0644)
52   if err != nil {
53     return "", err
54   }
55   defer idx.Close()
56   tpl := NewTmpl()
57   if err = tpl.Init(); err != nil {
58     return "", err
59   }
60   for _, name := range names {
61     tpl.AddPhoto(name)
62   }
63   tpl.RenderPage(idx, "photos.html")
64   return link, nil
65 }
66 func processFile(fh *multipart.FileHeader, dir
     string) (string, error) {
67   file, err := fh.Open()
68   if err != nil {
69     return "", err
70   }
71   defer file.Close()
72   fileName := fh.Filename
73   savePath := path.Join(dir,
     sanitizeFileName(fileName))
74   outFile, err := os.Create(savePath)
75   if err != nil {
76     return "", err
77   }
78   defer outFile.Close()
79   if _, err = io.Copy(outFile, file); err != nil
     {
80     return "", err
81   }
82   if err = scaleJPG(savePath); err != nil {
83     return "", err
84   }
85   return fileName, nil
86 }
87 func sanitizeFileName(fileName string) string {
88   allowed := regexp.
     MustCompile(`[^a-zA-Z0-9_\-\.]+`)
89   return allowed.ReplaceAllString(fileName, "")
90 }
```

Regex ohne Perl

Zeile 42 in Listing 1 muss aus dem Pfad zum Upload-Verzeichnis die letzten zwei Teilstücke extrahieren, um einen Link auf dem Webserver daraus zu generieren. Die letzten zwei Teilstrecken aus einem Pfad wie `/foo/bar/baz/` herauszuholen, ließe sich in Perl mit sogenannten Non-greedy Matches erledigen. Während `.*/` zum Beispiel immer gierig sämtliche Zeichen eines Strings bis zum letzten / aufschnappt (einschließlich weiterer Slashes innerhalb des Strings), begnügt sich dagegen `.*?/` mit allen Zeichen bis zum ersten Querstrich. Gos Regex-Engine implementiert allerdings nicht den vollen PCRE-Standard, und so hangelt sich Zeile 42 mittels `/[^/]+/[^/]+$` jeweils über die nächsten Nicht-Querstriche von einem Querstrich zum nächsten.

Zwei Ansätze

Bei dynamisch aufbereiteten Webseiten gehen die Meinungen auseinander: Die Jünger von PHP setzen auf funktions-

Listing 2: tmpl.go

```
01 package main
02 import (
03   "io"
04   tmpl "text/template"
05   "time"
06 )
07 type Photo struct {
08   Path  string
09   Thumb string
10 }
11 type TmplData struct {
12   CGI        string
13   TmplEngine *tmpl.Template
14   Date       string
15   OgDesc     string
16   OgImage    string
17   Photos     []Photo
18   Link       string
19 }
20 func NewTmpl() *TmplData {
21   return &TmplData{}
22 }
23 func (td *TmplData) Init() error {
24   td.Date = time.Now().Format("2006-01-02")
25   td.OgDesc = "Photos uploaded"
26   td.CGI = "/cgi/photoup"
27   te, err := tmpl.ParseGlob("tmpl/*.html")
28   td.TmplEngine = te
29   return err
30 }
31 func (td *TmplData) AddPhoto(name string) {
32   td.Photos = append(td.Photos, Photo{Path: name,
   Thumb: thumbName(name)})
33 }
34 func (td *TmplData) RenderPage(w io.Writer,
   tmplName string) error {
35   if len(td.Photos) != 0 {
36     td.OgImage = td.Photos[0].Thumb
37   }
38   for _, tpl := range []string{"intro.html",
   tmplName, "outro.html"} {
39     err := td.TmplEngine.ExecuteTemplate(w, tpl,
   td)
40     if err != nil {
41       return err
42     }
43   }
44   return nil
45 }
```

Listing 3: intro.html

```
01 <!DOCTYPE html>
02 <html lang="en">
03 <head>
04   <meta name="og:title" content="Photoup">
05   <meta property="og:description" content="{{.
   OgDesc}}">
06   <meta property="og:image" content="{{.
   OgImage}}">
07   <meta charset="UTF-8">
08   <meta name="viewport" content="width=device
   -width, initial-scale=1.0">
09   <link rel="stylesheet" href="/style.css">
10 </head>
11 <body>
12   <nav class="navbar">
13     <a href="../index.html">Photoup</a>  
        
14     <a href="#foobar">{{.Date}}</a>
15   </nav>
```

reiche Programmiersprachen, die innerhalb des HTML-Codes bei der Auslieferung auf Veranlassung des Servers Code ausführen, der innerhalb der HTML-Tags Ausgaben produziert. Im anderen Lager sitzen die Template-Anhänger mit Aufbereitungsprogrammen, die Vorlagen aus einem HTML-Gerüst mit Inhalt füllen. Dazu besetzen sie einfache Template-Variablen im Layout mit aktuellen Werten. Das Ergebnis speichern sie als statisches HTML ab, das der Server dann entsprechend schneller ausliefert.

Ob lieber ein Programm ein Template mit Variablen füllen soll oder ob das Template von sich aus Code startet, der die Lücken füllt, ist letztlich Geschmacksache. Im ersten Fall bleibt die Programmlogik innerhalb des Layouts einfach. Template-Engines bieten bewusst nur eingeschränkte Textersetzung. Vielleicht noch eine kleine For-Schleife, um Listen darzustellen, aber das war es dann auch schon: Schließlich sollen Code und Layout getrennt bleiben.

Kein Schnickschnack

Go bietet in seiner Standard-Library das Paket *text/template*, das typische Template-Aufgaben wie Textersatz effizient löst. Listing 2 verpackt die Template-Engine in ein objektorientiertes Paket mit einem Konstruktor und mit einer `Init()`-Methode, die ein Verzeichnis nach Templates durchsucht und diese anschließend für später aufgabelt.

So liegen sämtliche HTML-Snippets für das Layout der an den Browser zurückgeschickten Webseiten im Unterverzeichnis `tmpl/` bereit, und die Funktion `ParseGlob()` in Zeile 27 von Listing 2 liest alle dort gefundenen Dateien mit der Endung `.html` ein. Die Snippets in Listing 3 bis Listing 7 zeigen die Templates, die Platzhalter in dem Format `{{.Variable}}` enthalten. Diese ersetzt die Template-Engine dann mit den aktuellen Werten gleichnamiger Felder einer der Engine zur Laufzeit überreichten Struktur.

Lediglich das Snippet mit den Fotos in Listing 4 verwendet eine Funktion des *template*-Pakets, die über eine reine Variableninterpolation hinausgeht. Die Funktion `range()` ab Zeile 2 iteriert über die Variable `Photos`, die ein Array-Slice mit `Photo`-Strukturen erhält. Die einzelnen Elemente bieten ihrerseits entsprechend der Strukturdefinition ab der Zeile 7 in Listing 2 die Felder `Path` und `Thumb`. Die letzteren zwei Felder referenziert dann der HTML-Anker entsprechend.

Wenn die Engine ein Template vom Stapel lassen soll, nimmt die Methode `Execute()` des *template*-Pakets eine Struktur entgegen, die in ihren Feldern aktuelle Werte für die Template-Variablen enthält. Als weiterer Parameter dient ein Writer-Objekt, in das die Engine das Ergebnis schreibt. Diese Methode erwartet jedoch nur den einfachen Dateinamen (zum Beispiel `intro.html`) und nicht den vollständigen Pfad, um das gewünschte Template aus dem zuvor per `ParseGlob()` eingelesenen Satz zu extrahieren.

Vollkommen gleichgültig, ob das darzustellende Template nun `upload.html` oder `photos.html` heißt, umrahmt die Zeile 38 in Listing 2 es mit `intro.html` und `outro.html`, damit auch schönes HTML mit Anfang und Ende dabei herauskommt. Von `outro.html` stammt hier lediglich das abschließende `/body`-Tag.

Listing 4: photos.html

```
01   <div class="photo-gallery">
02   {{range .Photos}}
03   <div class='photo'> <a HREF={{.Path}}><img src={{.Thumb}}></
     img></a></div>\n
04   {{end}}
05   </div>
```

Listing 5: upload.html

```
01 <BR>
02 <form enctype="multipart/form-data" action="{{.CGI}}" method="post">
03   <input type="file" name="files" multiple>
04   <input type="submit" value="Upload">
05 </form>
```

Listing 6: done.html

```
<P>
  Upload complete. <A HREF="{{.Link}}">Share this link</A>
```

Vorschau aktivieren

Beim Blick auf das Template in Listing 3 fallen im HTML-Header eine Reihe ungewöhnlicher Tags auf, beispielsweise `og:title` oder `og:image`. Des Rätsels Lösung: Versendet man einen Link über die Apps iMessage oder Whatsapp, dann zeigt der Client manchmal eine Vorschau an [6]. Bei Links zu Artikeln in Nachrichtenmagazinen ist das oft ein kleines Bildchen, kombiniert mit der Schlagzeile sowie einigen Zeilen des Texts als Teaser für die Leser. Darauf klickt der Gesprächsteilnehmer dann neugierig, während er bloße Links häufig übersieht oder in unserer kurzatmigen Ära aus Zeitgründen gleich ganz ignoriert. Doch bei manchen Webseiten kommt keinerlei Vorschau. Bei anderen funktioniert es manchmal und manchmal nicht, was vom Tempo der Serverantwort abzuhängen scheint. Was ist da los, wie funktioniert dieses Feature denn eigentlich genau?

Um die Vorschau anzuzeigen, folgt der Messaging-Client dem Link, oft schon bevor der Sender *Send* drückt, und holt von der dortigen Website einige nach dem Open-Graph-Protokoll definierte Daten ein. Diesen Meta-Standard für Webseiten hatte ursprünglich die Firma Facebook auf den Weg gebracht. Er hat dazu gedient, das Internet zu archivieren

und Webseiten anhand von indizierten Metainformationen zu katalogisieren.

Drei Meta-Tags aus diesem Standard greifen nun typische Messaging-Clients im HTML-Code der gelinkten Webseite auf. Sie verwenden diese Informationen wiederum dazu, eine Vorschau anzuzeigen. Dabei entspricht `"og:title"` dem Kurztitel der Webseite, `"og:description"` bietet eine einzeilige Zusammenfassung des Inhalts, und `"og:image"` enthält eine URL auf eine JPEG-Datei, die die Vorschau dann bebildern soll.

Dabei müssen Webseitenbetreiber einiges beachten, damit Links auf ihren Inhalt kompatibel mit den Vorschauregeln bleiben. Da kein Standard konkret ausformuliert, was der Server machen muss, hilft lediglich stetiges Testen mit sämtlichen gängigen Messaging-Clients. Zum Beispiel darf keiner der beiden Text-Tags eine bestimmte Länge überschreiten, und auch die Größe des Meta-Bildchens im JPEG-Format ist beschränkt. Überdies muss die Webseite entsprechend zügig antworten, sonst bockt der Messenger und zeigt nur den Link ohne Info an.

Bildverarbeitung

Eine der ungeschriebenen Regeln der `og:`-Tags für die Vorschau lautet, dass das mit `og:image` referenzierte Foto nicht

Listing 7: image.go

```
01 package main
02 import (
03   "path/filepath"
04   "strings"
05   "github.com/disintegration/imaging"
06 )
07 func thumbName(fileName string) string {
08   ext := filepath.Ext(fileName)
09   name := strings.TrimSuffix(fileName, ext)
10   return name + "_s" + ext
11 }
12 func scaleJPG(inputPath string) error {
13   outputPath := thumbName(inputPath)
14   img, err := imaging.Open(inputPath, imaging.
AutoOrientation(true))
15   if err != nil {
16     return err
17   }
18   width := img.Bounds().Dx() / 4
19   height := img.Bounds().Dy() / 4
20   thumbnail := imaging.Thumbnail(img, width,
height, imaging.Lanczos)
21   err = imaging.Save(thumbnail, outputPath)
22   return err
23 }
```

Listing 8: util.go

```
01 package main
02 import (
03   "crypto/rand"
04   "crypto/sha256"
05   "encoding/hex"
06   "fmt"
07   "os"
08   "path"
09 )
10 func uploadDir() (string, error) {
11   root := os.Getenv("DOCUMENT_ROOT")
12   if root == "" {
13     return "", fmt.Errorf("No docroot")
14   }
15   randomStr := make([]byte, 32)
16   if _, err := rand.Read(randomStr); err != nil {
17     return "", err
18   }
19   hash := sha256.Sum256(randomStr)
20   shaDir := hex.EncodeToString(hash[:])
21   dir := path.Join(root, "uploads")
22   if _, err := os.Stat(dir); os.IsNotExist(err) {
23     if err := os.MkdirAll(dir, 0755); err != nil {
24       return "", err
25     }
26     err := os.WriteFile(path.Join(dir,
".htaccess"), []byte("Options -Indexes\n"), 0644)
27     if err != nil {
28       return "", err
29     }
30   }
31   path := path.Join(dir, shaDir)
32   err := os.MkdirAll(path, 0755)
33   if err != nil {
34     return "", err
35   }
36   return path, nil
37 }
```

6 **Vorschau eines Links in einem Whatsapp-Chat.**

größer als 1200 x 630 Pixel sein darf. Darum muss das CGI nach dem Hochladen aus den üblicherweise größeren Handyfotos (4032 x 3024 Pixel beim iPhone 12 Mini) ein kleines Vorschaubild generieren. Erschwerend kommt hinzu, dass Smartphones Fotos oft rotiert abspeichern und die Rotation im EXIF-Header der JPEG-Datei vermerken. Sie bürden es also der darstellenden Applikation auf, das Bild bei der Darstellung zu drehen.

Die Funktion `scaleJPG()` in Listing 7 holt sich darum das Paket *imaging* von Github: Es bietet die für uns sehr praktische Funktion `Thumbnail()` für JPEG-Fotos an und liest mit dem Setting `AutoOrientation(true)` (Zeile 14) das Foto gleich noch rotiert ein, falls der EXIF-Header das angibt. So stehen die Daten des Thumbnails für den Kontaktabzug korrekt in der Datei, und der Messaging-Client stellt das Bild richtig herum dar.

Im Dateinamen des verkleinerten Fotos wählt die Funktion `thumbName()` daraufhin ab Zeile 7 die Extension `_s`, und die Vorschau mit `og:image` referenziert das Bild im Template in Listing 3 mit diesem neuen Namen.

Schwer zu erraten

Unter welchem Verzeichnis ein Foto-Set auf dem Webserver erscheint, sollte ausschließlich der Empfänger des Links wissen, folglich muss der Server zufällige Namen erzeugen, die nur schwer zu erraten sind. Wie das URL-Feld des Browsers 4 zeigt, legt das CGI-Programm einen Kontaktabzug sämtlicher Fotos eines Sets unter einem Verzeichnis ab, das aus einem 64 Zeichen langen Hex-String besteht. Diesen zu erraten ist in etwa so schwer zu bewerkstelligen, wie ein Bitcoin zu schürfen!

Die Funktion `uploadDir()` in Listing 8 nutzt einen Zufallsgenerator sowie das Paket *crypto/sha256*, um einen SHA-256-Hash im Hex-Format zu erzeugen. Damit niemand auf die Idee kommt, direkt beim Webserver anzufragen und alle Verzeichnisse unter `uploads` aufzulisten, pflanzt Zeile 26 eine `.htaccess`-Datei ins Top-Verzeichnis und weist den Webserver mit `Options -Indexes` an, auf Anfragen zum Top-Verzeichnis mit *Permission Denied* zu antworten. Die einzelnen Unterverzeichnisse im SHA-256-Format hingegen liefert der Server klaglos aus.

Flugs zusammengebaut

Das fertige CGI-Binary erzeugt die Kommandosequenz in Listing 9 aus den Sourcen. Da der Hoster unter Umständen eine andere Plattform fährt als das Entwicklungssystem, stellt `GOOS=linux GOARCH=386` sicher, dass das Executable bei meinem Hoster auf Linux und einem Intel-Prozessor läuft. Dann noch das Binary ins CGI-Verzeichnis des Servers kopieren und sicherstellen, dass es sich ausführen lässt, schließlich mittels Rsync alle Templates ins Verzeichnis `tmpl/` direkt unterhalb kopieren, und es kann losgehen. Die Verzeichnisse `uploads/` und die SHA-1-Directories erzeugt das Programm zur Laufzeit selbstständig.

Listing 9: build.sh

```
$ go mod init photoup
$ go mod tidy
$ GOOS=linux GOARCH=386 go build photoup.go tmpl.go image.go util.go
```

Go-CGI: Pro und Kontra

Ein monolithisches CGI-Programm in Go hat Vor- und Nachteile. Dass der Server bei jedem Aufruf ein dickes Binary startet, zehrt klar an der Performance. Mehr als ein paar Hundert Aufrufe pro Tag sollte man dem Server nicht zumuten. Zudem stellt ein solches Binary mit recht viel Code einschließlich eingebundener Bibliotheken ein Sicherheitsrisiko dar, da es frei im Internet steht und jeder Schurke darauf herumorgeln darf. Enthielte eine Library einen sicherheitsrelevanten Bug, käme ein Update eines Servers nicht beim statisch kompilierten Binary an, es sei denn, es würde aktiv neu kompiliert.

Dass aber alles aus einem Guss ist und das Binary keine dynamischen Libraries (außer vielleicht der Libc) heranzieht, stellt sich allerdings dann als unschlagbarer Vorteil heraus, wenn der Serverbetreiber Aktualisierungen am Betriebssystem vornimmt. Derartige Wartungsarbeiten bringen Skripts oder dynamisch kompilierte Programme gern ins Stolpern, zum Beispiel wenn eine Bibliothek beim Upgrade von Ubuntu auf eine neuere Version plötzlich nicht mehr kompatibel ist. Ein statisches Binary läuft hingegen bis ans Ende der Zeit immer stur weiter.

Nicht Hinz und Kunz

Für den Produktionsbetrieb gilt es zu beachten, dass das Upload-Skript nicht frei im Internet herumstehen sollte. Sonst könnten Hinz und Kunz Fotos hochladen und allerlei Schabernack treiben. Entsprechend sollte das CGI-Programm entweder über eine `.htaccess`-Datei nur autorisierte Nutzer zulassen oder das Programm selbst auf einem gültigen User-Token bestehen. Entsprechenden Code einzupflanzen, fällt nicht schwer. (*uba*)

Dateien zum Artikel herunterladen unter
www.lm-online.de/dl/50139

Weitere Infos und interessante Links
www.lm-online.de/qr/50139