Körpergewicht im Blick mit CGI-Skript und Go

Abwägungssache

Mike Schilli steigt jede Woche auf die Waage und erfasst seine Gewichtsschwankungen als Zeitreihe. Dabei hilft ihm ein CGI-Skript in Go, das auch noch schöne Charts malt.

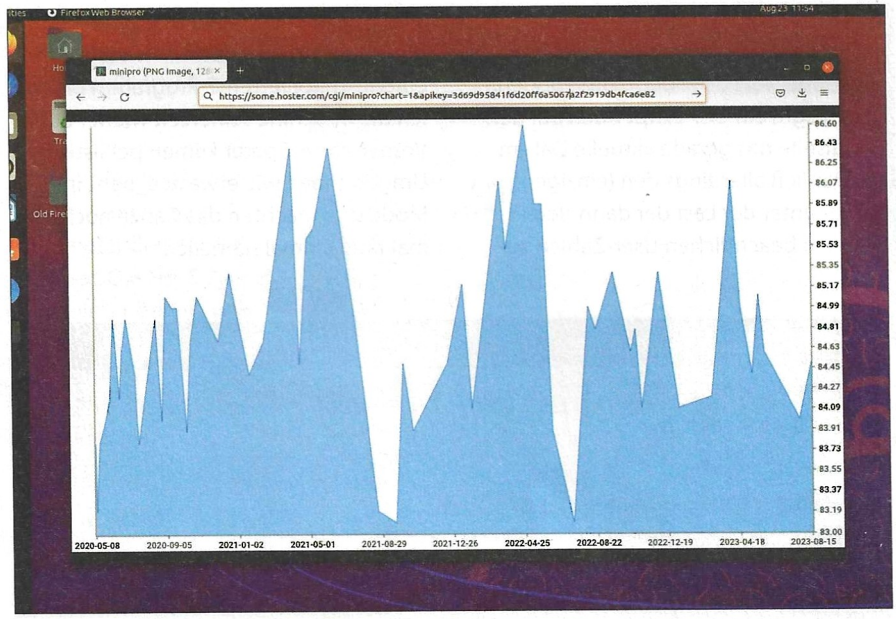
Mike Schilli

Datenpunkte aus Zeitreihen zu erfassen und grafisch aufzumöbeln, ist üblicherweise die Domäne von Tools wie Prometheus. Die Überwachungssoftware holt sich dafür in regelmäßigen Abständen den Status überwachter Systeme ab und speichert die Daten als Zeitreihe. Wenn Ausreißer auffallen, schlägt der Götterbote Alarm. Mit Darstellungswerkzeugen wie Grafana lassen sich die gesammelten Zeitreihen auf Wunsch in Dashboards über die letzte Woche oder das letzte Jahr verteilt als Graphen darstellen, so-

dass auch das höhere Management auf einen Blick weiß, wo der Hase langläuft.

Allerdings erlaubt mein Billigheimer-Hoster es mir nicht, auf meinem gemieteten virtuellen Server nach Belieben Softwarepakete zu diesem Zweck zu installieren. Auch wäre mir die Wartung derartig komplizierter Produkte mit ihren ständig fälligen Updates zu umständlich. Doch es gibt auf dem Webserver eine CGI-Schnittstelle aus den Neunzigerjahren. Wie schwer wäre es wohl, in Go ein CGI-Programm zu schreiben, das im Sinne einer API gemessene Werte per HTTPS entgegennimmt und die daraus erstellte Zeitreihe, übersichtlich als Grafik formatiert, im PNG-Format zum Browser zurückschickt?

Abbildung 1 zeigt den Graphen einer Zeitreihe, die mein Gewicht in Kilogramm über die vergangenen Jahre (für den Abdruck möglicherweise geschönt) als Grafik ausgibt. Das gleiche CGI-Skript nimmt auch neue Daten entgegen. Zeigt meine Waage zum Beispiel eines Tages 82,5 Kilogramm an, genügt der Aufruf `curl '.../cgi/minipro?add=82.5&apikey=Key'` um den Wert unter dem aktuellen Datum in die nun auf dem Server verwaltete Zeitreihe einzuspeisen. Ersetze ich in der URL `add=...` durch `chart=1`, kommt die Grafik aller bislang ermittelten und eingespeisten Werte zurück.



1 Die Gewichtsschwankungen des Autors über die Jahre.

Jurassic Tech

Dabei ist das CGI-Protokoll echte Saurier-technologie aus den 90er-Jahren des vergangenen Jahrhunderts. Damals kamen die ersten dynamischen Webseiten in Mode, nachdem die User, auf den Geschmack gekommen, nach mehr als statischem HTML zu lechzen begannen.

Ich erinnere mich noch ganz genau an diese Zeit: Damals arbeitete ich bei AOL, deren Webauftritt auf Aol.com ich als frischgebackener Importingenieur aus

Listing 1: `cgi-test.go`

```
01 package main
02 import (
03   "fmt"
04   "net/http"
05   "net/http/cgi"
06 )
07 func main() {
08   handler := func(w http.ResponseWriter, r *http.
   Request) {
09     qp := r.URL.Query()
10     fmt.Fprintf(w, "Hello\n")
11     for key, val := range qp {
12       fmt.Fprintf(w, "key=%s=%s\n", key, val)
13     }
14   }
15   cgi.Serve(http.HandlerFunc(handler))
16 }
```

Listing 2: Dockerfile

```
01 FROM ubuntu:18.04
02 ENV DEBIAN_FRONTEND noninteractive
03 RUN apt-get update
04 RUN apt-get install -y curl
05 RUN apt-get install -y vim make
06 RUN apt-get install -y git
07 RUN curl https://dl.google.com/go/go1.21.0.linux
   -amd64.tar.gz >go1.21.0.linux-amd64.tar.gz
08 RUN tar -C /usr/local -xzf go1.21.0.linux-amd64.
   tar.gz
09 ENV PATH="${PATH}:/usr/local/go/bin"
10 WORKDIR /build
11 COPY *.go *.mod *.sum /build
12 RUN go mod tidy
```

Germany die Ehre hatte aufzufrischen – damals live auf einem einzigen Server und ohne Netz oder doppelten Boden. Dort zeigte ein CGI-Skript oben auf der Portalseite das gerade aktuelle Datum an. Das ließ allerdings den (einzigen!) Server unter der Last der dann doch ziemlich beachtlichen User-Zahlen zusammenkrachen, weil bei jedem Aufruf ein Perl-Interpreter starten musste. Mit einem kompilierten C-Programm brachte ich die Maschine seinerzeit wieder auf Vordermann. Später kamen persistente Umgebungen wie etwa `mod_perl` in Mode und machten das Ganze noch einmal tausendmal schneller.

Listing 3: `Makefile.cgi-test`

```
01 DOCKER_TAG=cgi-test
02 SRCS=cgi-test.go
03 BIN=cgi-test
04 REMOTE_PATH=some.hoster.com/dir/cgi
05 remote: $(SRCS)
06         docker run -v `pwd`:/build -it $(DOCKER_TAG) \
07         bash -c "go build $(SRCS)" && \
08         scp $(BIN) $(REMOTE_PATH)
09 docker:
10         docker build -t $(DOCKER_TAG) .
```

All inclusive

Heute ist CGI verpönt, weil ein Skript eventuell ein Sicherheitsloch im Server aufreißt und die Startup-Kosten eines externen Programms, das bei jedem eingehenden Request startet, immens ausfallen. Für mein Gewichtsbarometer, bei dem der Server vielleicht zwei Requests pro Tag erhält, ist das Design allerdings vertretbar. In einer Skriptsprache wie Python wäre das Ganze auch ruckzuck implementiert.

Allerdings empfand ich es als Herausforderung, beide Funktionen in ein statisches Go-Binary zu packen, das keinerlei Abhängigkeiten aufweist. Alle naselang mit Pip3 eine Python-Bibliothek zum Malen von Charts aufzufrischen, ist auch kein Leben. Einmal kompiliert – und gern auf einer anderen Plattform crosskompiliert – läuft ein statisch gelinktes Programm bis zum Sankt-Nimmerleins-Tag. Selbst wenn es dem Hoster einfiele, die

Listing 4: `minipro.go`

```
01 package main
02 import (
03   "fmt"
04   "net/http"
05   "net/http/cgi"
06   "regexp"
07 )
08 const CSVFile = "weight.csv"
09 const APIKeyRef = "3669d95841f6d20ff6a5067a2f2919
   db4fca6e82"
10 func main() {
11   handler := func(w http.ResponseWriter, r *http.
   Request) {
12     qp := r.URL.Query()
13     params := map[string]string{}
14     for key, val := range qp {
15       if len(val) > 0 {
16         params[key] = val[0]
17       }
18     }
19     apiKey := params["apikey"]
20     if apiKey != APIKeyRef {
21       fmt.Fprintf(w, "AUTH FAIL\n")
22       return
23     }
24     if len(params["chart"]) != 0 {
25       points, err := readFromCSV()
26       if err != nil {
27         panic(err)
28       }
29       chart := mkChart(points)
30       w.Write(chart)
31     } else if len(params["add"]) != 0 {
32       sane, _ := regexp.MatchString(`^[.\d]+$`,
   params["add"])
33       if !sane {
34         fmt.Fprintf(w, "Invalid\n")
35         return
36       }
37       err := addToCSV(params["add"])
38       if err == nil {
39         fmt.Fprintf(w, "OK\n")
40       } else {
41         fmt.Fprintf(w, "NOT OK (%s)\n", err)
42       }
43     }
44   }
45   cgi.Serve(http.HandlerFunc(handler))
46 }
```

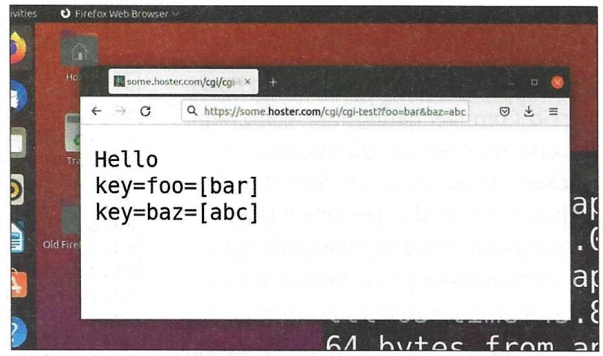
Linux-Distro auf eine neue Version anzuheben und irgendwelche Bibliotheken plötzlich verschwänden, liefe das All-inclusive-Go-Binary immer noch.

CGI zum Warmwerden

Stellt ein Webserver fest, dass er einen Request aufgrund seiner Konfiguration mit einem externen CGI-Skript beantworten muss, setzt er unter anderem die Umgebungsvariable `REQUEST_URI` auf die URL des Requests und ruft das Programm oder Skript auf. Letzteres holt dann die zur Bearbeitung des Requests notwendigen Informationen aus den Umgebungsvariablen. Bei einem GET-Request genügt etwa die URL in `REQUEST_URI`, deren Pfad auch alle CGI-Form-Parameter einschließt. Die Antwort schreibt das Skript einfach per `print()` nach Stdout. Den Textstrom greift der Webserver ab und schickt ihn an den anfragenden Client zurück.

Listing 1 zeigt ein minimales CGI-Programm in Go. Es verwendet die Standardbibliothek *net/http/cgi*, deren Funktion `Serve()` in Zeile 15 den eingehenden Request analysiert und dann die Antwort an den Server zurückschickt.

Dazu nimmt sie als Parameter eine Handler-Funktion entgegen (ab Zeile 8), die einen Writer für die Ausgabe und einen Reader für die Request-Daten als Parameter akzeptiert. Der Aufruf der Library-Funktion `Query()` auf die hereingereichte Request-URL gibt eine Map zurück, die die Namen der ankommenden CGI-Parameter den Werten zuweist. Die For-Schleife ab Zeile 11 iteriert über die Einträge in der Hashmap und gibt deren Namen und Werte jeweils an den Writer `w` aus.



2 Das Go-Programm aus Listing 1 als CGI-Skript in Aktion.

Für immer statisch

Kompiliert und gelinkt entsteht aus Listing 1 ein Binary, das man ausführbar ins Verzeichnis `cgi/` des Webservers kopiert. Der ist so konfiguriert, dass er bei einem eingehenden Request auf `cgi/cgi-test` das Programm `cgi-test` aufruft und dessen Ausgabe an den Browser des anfragenden Webclients zurückschickt. Abbildung 2 zeigt das Ergebnis aus Sicht des mittels eines Webbrowsers anfragenden Benutzers.

So weit, so gut – aber wie kompiliert man nun Listing 1? Schließlich soll dabei ein Binary herauskommen, das auf der Linux-Distro des Hosters läuft, und die ist unter Umständen inkompatibel zur Build-Umgebung, da ihr Shared Libraries in bestimmten, vielleicht mittlerweile veralteten Versionen fehlen. Go-Binaries benötigen normalerweise nur die Libc des Hostsystems in einer akzeptablen Version. Zu Hilfe kommt Docker. Mein Hoster nutzt Ubuntu 18.04, also startet das Dockerfile in Listing 2 die Umgebung mit diesem Basis-Image.

Allerdings hinkt die Version des Pakets *golang* notorisch hinterher; bei einem schon in die Jahre gekommenen Ubuntu ist sie nicht zu gebrauchen. Also zieht das Dockerfile in Zeile 7 den Tarball des aktuellen Go 1.21 aus dem Netz und pflanzt dessen Inhalt ins Root-Verzeichnis. Hinzu kommen noch einige Tools wie Git (Go nutzt Git zum Einholen von Github-Paketen) und Make für den Build, und fertig ist die Frankenstein-Distro.

Listing 5: `csv.go`

```
01 package main
02 import (
03   "encoding/csv"
04   "fmt"
05   "os"
06   "time"
07 )
08 func addToCSV(val string) error {
09   f, err := os.OpenFile(CSVFile,
10     os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
11   if err != nil {
12     return err
13   }
14   defer f.Close()
15   _, err = fmt.Fprintf(f, "%s,%d\n", val, time.
   Now().Unix())
16   return err
17 }
18 func readFromCSV() ([][]string, error) {
19   points := [][]string{}
20   file, err := os.Open(CSVFile)
21   if err != nil {
22     if os.IsNotExist(err) {
23       return points, nil
24     } else {
25       return points, err
26     }
27   }
28   defer file.Close()
29   reader := csv.NewReader(file)
30   points, err = reader.ReadAll()
31   return points, err
32 }
```

Gut vorbereitet

Zum Kompilieren von Go-Sourcen muss der Go-Compiler häufig Pakete aus dem Netz nachziehen und übersetzen. Ein Docker-Image, das diesen Schritt noch nicht enthält, trödelt bei jedem Übersetzungslauf teilweise minutenlang in der Vorbereitungsphase herum, die es bei jeder kleinen Änderung am Source-Code wieder und wieder ausführt. Zur Beschleunigung dieser Phase kopiert Zeile 11 in Listing 2 die Go-Quellen und die Moduldateien in das Image, und `go mod tidy` in Zeile 12 kompiliert schon einmal alles vor. Startet anschließend ein Container basierend auf dem Image, muss Go nur noch die Sourcen lokal übersetzen und alles zusammenlinken, was in Sekunden passiert. So machen das Entwickeln und die Fehlersuche wieder Spaß.

Das Makefile aus Listing 3 baut unter dem Target `docker` (ab Zeile 9) das Image und weist ihm das Tag `cgi-test` zu. Zum Kompilieren des Codes rufen Sie das Target `remote` auf (ab Zeile 5), das mit `docker run` einen Container startet und das Build-Verzeichnis im Container auf das aktuelle Verzeichnis auf dem Host einhängt. So ist das Binary auch außerhalb des Containers verfügbar.

```
$ tail -20 weight.csv
85.9,1652400000
85.9,1653523200
83.9,1654819200
83.1,1657756800
85,1660003200
84.8,1661040000
85.3,1663545600
84.6,1666051200
84.8,1666828800
84.1,1667692800
85.3,1670198400
84.1,1673049600
84.2,1677715200
86.1,1680739200
85,1681948800
84.4,1683590400
85.1,1684540800
84.6,1685318400
84,1690329600
84.5,1692144000
$
```

3 Die Wiegemessungen als Fließkommawerte mit Zeitstempel.

Den eigentlichen Build-Prozess übernimmt das Shell-Kommando in Zeile 7, das `go build` aufruft. Klappt das fehlerfrei, findet eine Secure Shell mit Scp das Endprodukt im aktuellen Verzeichnis (aber außerhalb des Containers) und installiert es auf dem Ziel-Host. Dessen Adresse gibt Zeile 4 mit `REMOTE_PATH` vor.

Kein Pillepalle

Nun aber genug mit dem Pillepalle. Das eigentliche CGI-Programm, das neue Werte für die Zeitreihe entgegennimmt und später grafisch anzeigt, heißt `minipro` und steht in Listing 4. Es nimmt neue Wiegemessungen über die CGI-Schnittstelle mit dem Parameter `add` vom Benutzer entgegen und legt sie auf dem Server unter dem Zeitstempel der aktuellen Uhrzeit in der CSV-Datei `weight.csv` ab. Das erledigt die Funktion `addToCSV()` ab Zeile 37.

Damit nicht Hinz und Kunz auf die Schnittstelle zugreifen kann, fordert das CGI-Programm einen API-Key, der hart-

Listing 6: `chart.go`

```
01 package main
02 import (
03   "bytes"
04   "github.com/wcharczuk/go-chart/v2"
05   "strconv"
06   "time"
07 )
08 func mkChart(points [][]string) []byte {
09   xVals := []time.Time{}
10   yVals := []float64{}
11   header := true
12   for _, point := range points {
13     if header {
14       header = false
15       continue
16     }
17     val, err := strconv.ParseFloat(point[0], 64)
18     if err != nil {
19       panic(err)
20     }
21     added, err := strconv.ParseInt(point[1], 10, 64)
22     if err != nil {
23       panic(err)
24     }
25     xVals = append(xVals, time.Unix(added, 0))
26     yVals = append(yVals, val)
27   }
28   mainSeries := chart.TimeSeries{
29     Name: "data",
30     Style: chart.Style{
31       StrokeColor: chart.ColorBlue,
32       FillColor:   chart.ColorBlue.
   WithAlpha(100),
33     },
34     XValues: xVals,
35     YValues: yVals,
36   }
37   graph := chart.Chart{
38     Width:  1280,
39     Height: 720,
40     Series: []chart.Series{mainSeries},
41   }
42   w := bytes.NewBuffer([]byte{})
43   graph.Render(chart.PNG, w)
44   return w.Bytes()
45 }
```

kodiert in Zeile 9 steht. Der anfragende API-User legt ihn unter dem CGI-Parameter `apikey` dem Request bei. Das Programm bearbeitet den Request nur dann weiter, wenn der Schlüssel mit dem hartkodierten Wert übereinstimmt, sonst ist ab Zeile 21 Schluss.

Weil man CGI-Parametern allgemein nicht trauen kann, empfiehlt es sich, sie mit regulären Ausdrücken auf ihre Gültigkeit zu prüfen. So beschnuppert Zeile 32 den Parameter `add` darauf, ob der String tatsächlich wie eine Fließkommazahl aussieht, also ausschließlich aus Ziffern und Punkten besteht. Falls ja, steht die Variable `sane` auf `true`, falls nein, bricht Zeile 34 mit einer Fehlermeldung ab.

Gut geraten

Um ein Diagramm der Zeitreihe der bislang eingespeisten Werte anzusehen, setzen Sie den CGI-Parameter `chart` auf einen beliebigen Wert. Daraufhin reagiert der Abschnitt ab Zeile 24 in Listing 4, erzeugt mit `mkChart()` (siehe Listing 6) eine neue Chart-Datei im PNG-Format und gibt in Zeile 30 die Binärdaten der Grafik per `w.Write()` an den anfragenden Browser zurück. Glücklicherweise ist die Library *net/http/cgi* so schlau, dass sie den einleitenden HTTP-Header auf `Content-Type: image/png` setzt, wenn sie die ersten paar Bytes des Stroms untersucht und dort Sequenzen findet, die auf ein PNG-Image hindeuten.

Listing 5 übernimmt die Verwaltung der CSV-Datei. Deren Inhalt besteht aus den Fließkommawerten der Gewichtsmessungen, denen jeweils pro Zeile nach einem Komma ein Zeitstempel im Epoch-Format beiliegt. Abbildung 3 zeigt einen Teil der gespeicherten Daten.

Schreiben mit Garantie

Das Einspeisen von neuen Werten übernimmt in Listing 5 die Funktion `addToCSV()` ab Zeile 8. Sie öffnet die CSV-Datei im Modus `O_APPEND`, sodass die Schreibfunktion `fmt.Fprintf()` in Zeile 15 neue Werte mit anhängendem aktuellen Zeitstempel stets am Ende der Datei hinzufügt.

Dieser Modus hat einen angenehmen Nebeneffekt: Er sorgt dafür, dass unter Posix-kompatiblen Unix-Systemen Zeilen, die nicht länger als `PIPE_BUF` sind (unter Linux in der Regel 4048 Bytes), immer vollständig geschrieben werden, ohne dass ein anderer Prozess dazwischenfunken und die Zeile ruinieren kann. Im vorliegenden Fall ist das nicht so wichtig, da kaum Requests eintreffen, aber auf einem heftig schwitzenden Webserver wäre ohne diese Garantie (oder einen manuell gesetzten Lock) die Datei schnell korrupt.

Umgekehrt liest `readFromCSV()` ab Zeile 18 die Zeilen aus der CSV-Datei, und das Standardpaket `encoding/csv` fieselt die kommaseparierten Einträge auseinander. Auf diesem Weg kommt ein zweidimensionales Array-Slice von Strings zurück, mit zwei Einträgen pro Zeile für Wert und Zeitstempel.

Grafisch aufgehübscht

Diese Matrix mit Datenpunkten nimmt `mkChart()` ab Zeile 8 in Listing 6 entgegen und produziert daraus einen Graphen wie den aus Abbildung 1. Die Umrechnung der Zeitstempel aus dem Unix-Format in ein gut lesbares Format für die x-Achse übernimmt das Paket *go-chart* von Github automatisch. Zeile 4 in Listing 6 zieht es herein.

Zeile 28 erzeugt aus den Datenpunkten in `xVals` (Zeitstempel) und `yVals` (Gewichtsmessungen) eine Struktur vom Typ `chart.TimeSeries`. Die baut nun die Struktur `chart.Chart` ab Zeile 37 in ein Diagramm ein. Die Funktion `Render()` in Zeile 43 formt daraus die Binärdaten einer PNG-Datei mit dem Diagramm.

Dazu legt Zeile 42 in der Variablen `w` einen neuen Write-Puffer an, in den die Chart-Funktion hineinschreibt, und `Bytes()` in Zeile 44 gibt dessen rohe Bytes an den Aufrufer der Funktion, also an das Hauptprogramm, zurück.

Um die drei Quelldateien zu einem statischen Binary zusammenzufügen, baut das Makefile aus Listing 7 ähnlich wie vorher unter dem Target `docker` mit demselben Dockerfile ein neues Image mit dem Tag `minipro` zusammen. Ist das geschafft, startet `make remote` den Container, hängt zum späteren Einsammeln des fertigen Binarys dessen Arbeitsverzeichnis ein, und startet danach mit `go build` den Compile- und Link-Prozess.

Wenn das ohne Fehler klappt, kopiert die Secure Shell das Binary mit Scp in das CGI-Verzeichnis des in `REMOTE_PATH` eingestellten Hosters. Von dort kann ein Browser oder ein Curl-Skript anschließend die Funktionen abrufen, mit `add` neue Datenpunkte hinzufügen und mit `chart` den bestehenden Datensatz grafisch aufmöbeln und illustrieren. (*uba*)

Listing 7: `Makefile.build`

```
DOCKER_TAG=minipro
SRCS=minipro.go chart.go csv.go
BIN=minipro
REMOTE_PATH=some.hoster.com/dir/cgi
remote: $(SRCS)
        docker run -v `pwd`:/build -it $(DOCKER_TAG) \
        bash -c "go build $(SRCS)" && \
        scp $(BIN) $(REMOTE_PATH)
docker:
        docker build -t $(DOCKER_TAG) .
```

Dateien zum Artikel herunterladen unter
www.lm-online.de/dl/48757

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.