Ein Einblick in die Webentwicklung mit Google Go

Wünsch dir was mit Go

Die Innovationsmaschine Google hat mit Go eine gut gemachte Sprache für die Server- und Backend-Programmierung geschaffen, die mittlerweile etabliert ist. Sie ist eine wichtige Toolsprache für DevOps und für den Einsatz im Cloud-Computing und wird mit vielen Features für die Netzwerk- und Webprogrammierung ausgeliefert. Dieser Artikel stellt Go als Entwicklungsgrundlage für Webapplikationen vor.

von Daniel Stender

Das konsequent auf Effizienz getrimmte Go von Google (auch: „Golang") [1] überzeugt als eine Art moderneres C mit performanten, praktischen, statisch gelinkten Binaries und kurzer Kompilationszeit auch bei umfangreichen Projekten. Darüber hinaus hat die Sprache neben einer integrierten Toolchain und einer schlanken Syntax auch eine gradlinige, pragmatische Architektur und starke Features wie implizite Struct/Method-Interfaces und eine eingebaute Nebenläufigkeit. Die stark idiomatische, imperative Sprache mit objektorientierten Elementen steht mit starker Typisierung und Garbage Collection irgendwo im Dreieck zwischen C, Java und Skriptsprachen wie Python. Sie hat auf jeden Fall mittlerweile einen festen Platz in der System- und Backend-Programmierung. Fans sehen dabei pragmatisch über kleinere Mängel der Sprache hinweg. Seit 2012, als Go 1.0 veröffentlicht worden ist, hat sie als wichtige Toolsprache für DevOps die Rolle von Ruby übernommen, und auch im Cloud-Computing eine tragende Position erlangt. Die Liste der in Golang implementierten DevOps- und Cloud-Computing-Applikationen ist lang: Dazu gehören zum Beispiel Docker genauso wie Kubernetes, Prometheus, Elastic Beats, etcd, LXD, und HashiCorp-Tools wie Terraform, Packer, Consul und Vault. Eine Go Library für den programmgesteuerten Zugriff auf die eigene API für das zugehörige CLI-Tool sind auch bei mittelgroßen Cloudanbietern wie DigitalOcean und Hetzner Standard.

Go wird mit gut ausgestatteter und moderner Standardbibliothek geliefert, zu der zum Beispiel auch Kryptographiepakete gehören. Eine Reihe der eingebauten Pakete unterstützen die Webentwicklung, also das Aufsetzen von Applikationen, die von einem Browser bzw. über das HTTP-Protokoll angesprochen werden. Da gibt es zunächst das *http*-Paket mit einem robusten und produktionsreifen HTTP-Server, das *template*-Paket, das eine Sprache für HTML-Templates bietet, sowie das *json*-Paket mit dem Sie JSON-Dokumente in Go-Datenstrukturen ein- und auspacken können, um etwa RESTful APIs damit aufzusetzen. Go eignet sich von seinem Ansatz her gut, um Microservices zu implementieren. Die eingebauten Features für die Webprogrammierung sind mit Server-Logging und Authentication und anderen Dingen auch nützlich für die allgemeine Backend-Entwicklung.

Go steht unter der BSD-Lizenz und ist frei verfügbar und einsetzbar. Dieser Artikel führt als Tutorial in die zentralen Elemente der Webentwicklung mit dieser Sprache ein, und möchte dabei auch ihre Effektivität darstellen. Für das Verständnis grundlegender Sprachelemente sei hier aus Platzgründen auf die allgemeine

Einführungsliteratur verwiesen [2]. Auf einige allgemeine Basics werde ich aber noch eingehen.

„net/http“

Der HTTP-Server in der Stdlib von Golang [3] baut auf vier Elementen auf: ein Listener, ein Multiplexer, ein Ausgabekanal und ein Datenobjekt, das das eingehende HTTP Requests repräsentiert.

Ein einfaches „Hello-World“-Beispiel zeigt Listing 1. Die Funktion *http.ListenAndServe* blockt das Programm (STRG + C bricht es ab) und wartet dann auf eingehenden Request. Sie nimmt als Parameter den gewünschten Port für den Server. Mit *nil* wählen Sie hier den Default-Multiplexer aus. Es bietet sich an, dass Sie den Aufruf des Listeners, wie in Listing 1 gezeigt, in *fmt.Println* einbetten, das eine Fehlermeldung beim Programmabbruch erzeugt, wenn es Probleme gibt (wenn etwa der Port besetzt oder blockiert ist). Die Funktion *http.HandleFunc* fungiert als Multiplexer, der die in einem HTTP Request angeforderte Route (im Beispiel ist lediglich „/“ vorgesehen) mit einer Funktion verknüpft, die bei Auslösung dieses Endpunktes als Handler ausgeführt wird.

Für einen funktionierenden Handler sind zwei Parameter vorgeschrieben: *http.ResponseWriter* und *http.Request* (den Sie als Pointer ansprechen). Das Request-Datenobjekt kann auch unausgewertet bleiben, wie im ersten Beispiel. Den Ausgangskanal können Sie etwa mit *fmt.Fprintf* ansprechen, um so Text an den anfordernden Klienten abzusetzen. Die Go-Bibliothek überprüft die ersten Bytes der Rückgabe und gibt als *Content-Type* im Rückgabeheader ordentlich *text/html* zurück, wenn wie im Beispiel kein Plain Text, sondern HTML zurückgeschickt wird. Starten Sie das Beispiel mit *go run* und fragen Sie den dann auf *127.0.0.1:8000* laufenden Webserver zum Beispiel mit dem CLI-Tool *curl* ab. Überprüfen Sie die dabei zurückgegebenen Felder. Mit der Funktion *w.Header().Add* im Code ergänzen Sie noch das Datenfeld *Server*:

Der HTTP-Server in der Bibliothek Stdlib von Golang baut auf vier Elementen auf.

```
$ curl -i 127.0.0.1:8000
HTTP/1.1 200 OK
Server: Golang
Date: Fri, 30 Aug 2019 11:22:15 GMT
Content-Length: 38
Content-Type: text/html; charset=utf-8

<html><body>Hello, world!</body></html>
```

Kompilieren Sie den Quellcode mit *go build*, dann erhalten Sie als Webserver ein Binary, dass mit ca. 6,3 MB zugegebenermaßen nicht besonders schlank ist. Es enthält allerdings die beiden verwendeten Bibliotheken statisch gelinkt, den Garbage Collector und andere Dinge. Dass die erzeugten Binaries ohne irgendwelche Abhängigkeiten selbstständig lauffähig und damit hoch portierbar sind, ist dabei der Sinn der Sache. Auf einem Linux-Server wrappen Sie das Binary am besten in einen eigenen Systemd Service, um einen solchen Go-Webserver damit nach dem Booten automatisch zu starten.

Listing 2 zeigt, wie Sie das Request-Objekt *http.Request* für eine beliebige Go-Datenstruktur auswerten. Es enthält den angeforderten Pfad (abfragbar mit *URL.Path*) und andere Elemente. Mit der im Beispiel gezeig-

Listing 1

```
package main

import (
  "fmt"
  "net/http"
)

func myhandler(w http.ResponseWriter, r *http.Request) {
  w.Header().Add("Server", "Golang")
  fmt.Fprintf(w, "<html><body>Hello, world!</body></html>")
}

func main() {
  http.HandleFunc("/", myhandler)
  fmt.Println(http.ListenAndServe(":8000", nil))
}
```

Listing 2

```
type Person struct {
  Vorname  string
  Nachname string
}

func myhandler(w http.ResponseWriter, r *http.Request) {
  p1 := Person{Vorname: "Max", Nachname: "Mustermann"}
  if vname := r.URL.Query().Get("vorname"); vname != "" {
    p1.Vorname = vname
  }
  if nname := r.URL.Query().Get("nachname"); nname != "" {
    p1.Nachname = nname
  }
  s1 := fmt.Sprintf("<html><body>Hello, %s %s!</body></html>", p1.Vorname,
                                                              p1.Nachname)
  w.Write([]byte(s1))
}
```

ten Methode *URL.Query.Get* können Sie optionale Werte aus komplexen *HTTP-GET*-Anfragen extrahieren. Alternativ dazu gibt es die Methode *FormValue*. Dieser Webserver kann mit wenigen Zeilen Code mehr also bereits Usereingaben auswerten:

```
$ curl 127.0.0.1:8000
<html><body>Hello, Max Mustermann!</body></html>
```

Listing 3

```
func RootInfo(w http.ResponseWriter, r *http.Request) {
  http.Error(w, "Wählen Sie einen geeigneten Endpunkt!", 404)
}

func NameHandler(w http.ResponseWriter, r *http.Request) {
  params := mux.Vars(r)
  fmt.Fprintf(w, "Hello %s %s!", params["vorname"], params["nachname"])
}

func main() {
  mymux := mux.NewRouter()
  mymux.HandleFunc("/", RootInfo)
  mymux.HandleFunc("/{vorname}/{nachname}", NameHandler)
  fmt.Println(http.ListenAndServe(":8000", mymux))
}
```

Listing 4

```
var mytemplate string = "<html><body>Hallo {{.Vorname}} {{.Nachname}}</body></html>"

type MyContext struct {
  Vorname  string
  Nachname string
}

func UseStruct(w http.ResponseWriter, r *http.Request) {
  c1 := MyContext{"John", "Doe"}
  t1 := template.New("foo")
  t1.Parse(mytemplate)
  t1.Execute(w, c1)
}

func UseMap(w http.ResponseWriter, r *http.Request) {
  c2 := map[string](string){"Vorname": "John", "Nachname": "Doe"}
  t2 := template.New("bar")
  t2.Parse(mytemplate)
  t2.Execute(w, c2)
}

func main() {
  http.HandleFunc("/struct/", UseStruct)
  http.HandleFunc("/map/", UseMap)
  fmt.Println(http.ListenAndServe(":8000", nil))
}
```

```
$ curl "127.0.0.1:8000/?vorname=John&nachname=Doe"
<html><body>Hello, John Doe!</body></html>
```

Alternativ zu *fmt.Println* in Listing 1 verwendet dieser Code die Methode *Write* für die Rückgabe, die Byte-Arrays verarbeitet (was Strings in Go „unter der Haube" ja auch sind).

Multiplexer

Der in Golang eingebaute Multiplexer *ServeMux* (der etwa von der Convenience-Funktion *http.HandleFunc* angesprochen wird) hat gegenüber ausgewachsenen HTTP Frameworks in anderen Programmiersprachen (vergleichen Sie das etwa mit Flask [4] für Python) einige Beschränkungen: so können HTTP-Methoden wie *GET*, *PUT* usw. nicht abgefragt bzw. für bestimmte Endpunkte nicht vorgeschrieben werden. Zudem lassen sich die Elemente von Pfaden nicht als Variablen auswerten, und einige weitere Features fehlen hier. Diese Mängel lassen sich überwinden, indem Sie den alternativen Multiplexer *mux* aus dem Gorilla Web Toolkit [5] einsetzen, wie in Listing 3 gezeigt. Importieren Sie dafür zunächst einfach *github.com/gorilla/mux*. Um das Beispiel kompilieren zu können, müssen Sie das Paket vorher natürlich mit *go get* in Ihren *$GOPATH* ziehen.

Um das Paket zu verwenden, generieren Sie dann zunächst einen neuen Multiplexer bzw. Router mit NewRouter (Listing 3; der Name der dafür eingesetzten Variable ist dabei natürlich beliebig). Damit können Sie mit *HandleFunc* Pfade bzw. Endpunkte für diesen Router definieren und Funktionen verknüpfen. Außerdem benötigt *http.ListenAndServe* statt *nil* den Namen des angelegten Routers.

Die Verknüpfung zu dem in Listing 3 enthaltenen *NameHandler* wertet die beiden Elemente einer zweistelligen Route aus HTTP Request als Variablen aus. Diese Variablen können Sie mit *mux.Vars* abfragen, das eine Map (eine Reihe von Key/Value-Paaren) in der Form *map[string](string)* zurückgibt. Die Map *params* im gezeigten Beispiel können Sie dann mit *params[„foo"]* anhand der darin enthaltenen Keys (*vorname* und *nachname* aus dem Request) nach Bedarf auswerten.

Eine weitere Funktion *RootInfo* reagiert auf Anfragen auf dem Root-Pfad des Webservers. Die Funktion *http.Error* der Stdlib-Bibliothek gibt einen beliebigen Fehlertext mit einem HTTP-Returncode zurück, und der Benutzer wird darauf hingewiesen, dass er den Webserver anders ansprechen muss:

```
$ curl -i 127.0.0.1:8000
HTTP/1.1 404 Not Found
Content-Type: text/plain; charset=utf-8
{...}
Wählen Sie einen geeigneten Endpunkt!
$ curl 127.0.0.1:8000/John/Doe
Hello John Doe!
```

„html/template“

Die Stdlib von Go bietet mit *html/template* [6] eine eingebaute Engine, die Vorlagen für HTML-Seiten für den Webserver rendern kann (die Variante *text/template* ist nicht für HTML vorgesehen). Die Templatesprache dieser Bibliothek besteht dabei aus Elementen, die in doppelten geschweiften Klammern eingefasst sind. Eine einfache Methode für die Verarbeitung von Templates ist zunächst, dass Sie diese einfach mit in den Code von *main* einbetten, wie in Listing 4 dargestellt. Sie können mit beliebigen Kontexten, mit Structs, aber auch mit einer im Beispiel gezeigten Map ausgewertet werden.

Ein neues Template müssen Sie zunächst einmal mit *template.New* initialisieren. Dabei wird über die verwendete Variable hinaus ein Bezeichner benötigt, der für bestimmte Konstruktionen eine Rolle spielt (im Beispiel unerheblich und deshalb einfach *foo* und *bar*). Danach parsen Sie mit der Methode *Parse* das neue Template, und schreiben es mit *Execute* schließlich in den HTTP-Ausgangskanal. Im Beispiel werden dabei die beiden Datenfelder aus den Kontexten (Struct *MyContext* und eine Map mit Strings als Keys und Values) ausgewertet. Das Ergebnis ist für beide angelegten Endpunkte identisch:

```
$ curl 127.0.0.1:8000/struct/
<html><body>Hallo John Doe</body></html>
$ curl 127.0.0.1:8000/map/
<html><body>Hallo John Doe</body></html>
```

Listing 5

```
func main() {
  p1 := map[string](string){"Vorname": "Max", "Nachname": "Mustermann"}

  templates := template.Must(template.ParseFiles("index.html"))

  http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    if err := templates.ExecuteTemplate(w, "index.html", p1); err != nil {
      http.Error(w, err.Error(), http.StatusInternalServerError)
    }
  })

  fmt.Println(http.ListenAndServe(":8000", nil))
}
```

Listing 6

```
$ curl -i 127.0.0.1:8000
HTTP/1.1 200 OK
Content-Type: application/json
Date: Sat, 31 Aug 2019 12:51:13 GMT
Content-Length: 204

[{"künstler":"Peter Gabriel","titel":"So!","jahr":1986},{"künstler":"Entombed",
    "titel":"Clandestine","jahr":1991},{"künstler":"Maxwell","titel":"Embrya","jahr":1999}]
```

Mit eigenständigen Templatedateien kann das *template*-Paket natürlich auch umgehen, und diese für den schnelleren Zugriff ohne Festplattenarbeit sogar cachen. Das funktioniert aber nur, wenn Sie nicht das standardmäßige *ioutil.ReadFile* für das Laden von Templates benutzen, sondern dafür die Funktion *template.ParseFiles* einsetzen (Listing 5).

ParseFiles kann mehrere Dateinamen bzw. -pfade verarbeiten. Die eingelesenen Templates stehen dann unter ihren Dateinamen für die Verarbeitung bereit (siehe *index.html* bei *ExecuteTemplate*). Der Pfad ist relativ zu dem Ort, an dem *go run* ausgelöst wird, und relativ zum resultierenden Binary. Das Parsen als separater Schritt entfällt bei dem Einsatz von *ParseFiles*. Die Methode *template.Must* ist ein Convenience Wrapper, der eine Fehlermeldung zurückgibt, falls beim Laden und Parsen der Templates Probleme auftreten.

Der Code in Listing 5 ist ein wenig anders arrangiert. Es gibt zum Beispiel keine separate *Handler*-Funktion, was zunächst keine besondere Auswirkung hat. Eine weitere Abweichung im Gegensatz zu vorherigen Beispielen ist der Einsatz des *ExecuteTemplate*, das hier in einer *If*-Schleife steht. Die Funktion gibt (wie eingebauten Funktionen in Go generell) eine Fehlermeldung zurück, falls Schwierigkeiten bei der Verarbeitung auftreten. Die *If*-Schleife prüft, ob das der Fall ist, und gibt dann eine Fehlermeldung an den User zurück. Die Konstante *http.StatusInternalServerError* repräsentiert dabei den HTTP-Fehlercode 500. Sie können die Ausführung der Funktion und das Prüfen auf *err* auch in zwei separaten Schritten vornehmen, das ist bei Go aber eher selten und mehr eine Stilfrage.

Das hier gezeigte Beispiel zu Elementen der Templatesprache (die Ausgabe von Datenfeldern) kratzt wirklich nur ganz an der Oberfläche. Es sind hier auch Schleifen

Listing 7

```
type Album struct {
  Artist string `json:"künstler"`
  Title  string `json:"titel"`
  Year   int64  `json:"jahr"`
  rating float64
}

func DumpJSON(w http.ResponseWriter, r *http.Request) {
  a1 := []Album{
    Album{"Peter Gabriel", "So!", 1986, 9.5},
    Album{"Entombed", "Clandestine", 1991, 9.3},
    Album{"Maxwell", "Embrya", 1999, 9.1},
  }
  j1, _ := json.Marshal(a1)
  w.Header().Set("Content-Type", "application/json")
  w.Write(j1)
}
```

Die Stdlib von Go enthält ein json-Paket, mit dem sich JSON-Dokumente in Go-Datenstrukturen deserialisieren bzw. serialisieren lassen, was die Mittel von Go für die Webprogrammierung vervollständigt.

im Template möglich. Mit *range* können Sie komplexe Datenstrukturen iterieren, es lassen sich Funktionen aufrufen und vieles mehr. Die Funktionalität dieser Bibliothek steht hinter vergleichbaren Lösungen, wie zum Beispiel Jinja2 für Python, nicht zurück.

„encoding/json"

Die Stdlib von Go enthält unter *encoding/json* ein json-Paket [7], mit dem Sie JSON-Dokumente in Go-Datenstrukturen einlesen („auspacken", Deserialisierung, Unmarshalling) und daraus generieren („packen", Serialisieren, Marshalling) können. Die Bibliothek harmoniert mit den eingebauten Webtools. Sie arbeitet auch mit komplexen Datentypen wie Slices und Maps zusammen, besonders gut aber mit Structs. Die eingebaute JSON-Funktionalität vervollständigt die Mittel von Go für die Webprogrammierung. Dieses Format spielt zum Beispiel bei APIs eine große Rolle für den Datenaustausch im Internet. Eine einfache JSON-Serialisierung zeigt Listing 7.

Die Funktion *json.Marshal* analysiert die vorliegende Datenstruktur (im Beispiel in Listing 7 ein Slice von Structs) und bildet sie entsprechend in JSON mit den korrespondierenden Datentypen (String, Integer, Boolean) ab. Auch „genestete" Strukturen werden entsprechend übertragen. Private Datenfelder (die mit Kleinbuchstaben beginnen) werden nicht übermittelt. Sehr nützlich sind die *json*-Tags bei *Struct*-Bestandteilen, die die korrespondierenden JSON-Datenfelder angeben. Die Methode *Header().Set* können Sie verwenden, um den korrekten *Content-Type* für die Rückgabe mitzugeben (Listing 6).

Für die Ausgabe an den Ausgabekanal eignet sich die *Write*-Methode, da die Funktionen dieser Bibliothek mit Byte Slices als Ein- und Ausgabe arbeiten. Auf die Abfrage eines eventuellen Verarbeitungsfehlers von *json.Marshal* und anderen Funktionen können Sie verzichten, indem Sie wie in Listing 7 einen *blank identifier* („_") bei der Abfrage einsetzen (wenn Sie einen Variable wie *err* angeben, dann müssen Sie diese auch verwerten). Eine alternative Methode für die Serialisierung mit diesem Paket ist die Verwendung eines JSON-Encoders mit *json.NewEncoder* und zugehörigen Methoden.

Fazit

Die großen Internetkonzerne haben immer genügend Mittel, um völlig neue Werkzeuge für sich zu entwickeln, wenn die auf dem Markt verfügbaren Lösungen den eigenen hohen Anforderungen nicht ganz genügen. Oftmals werden die neuen Produkte dann irgendwann als Open-Source-Produkte der Community geschenkt. Genauso hat man bei Go nochmal bei null angefangen und eine ganz neue Programmiersprache für die Serverprogrammierung praktisch am Reißbrett entwickelt. Die Funktionen der zugehörigen Stdlib sind dabei nach dem „Wünsch-dir-was"-Prinzip vor dem Hintergrund des modernen Anforderungsprofils einer Internetfirma zusammengestellt worden. Dementsprechend findet der Programmierer hier eine Fülle von relevanten Bibliotheken, zu denen natürlich auch Pakete für die Webprogrammierung gehören. Go ist vollkommen auf Effektivität getrimmt, was sich auch bei den mit wenig Aufwand einsetzbaren Bibliotheken bemerkbar macht.

Daniel Stender ist freier DevOps Engineer und arbeitet unter anderem für ITP Nord (Potsdam) und Ratbacher (Stuttgart) als Techniker und Berater bei großen Kunden aus dem Bereich Banken und Finanzdienstleistung.

https://danielstender.com

Links & Literatur

[1] https://golang.org

[2] Hanna, Tam: „Einführung in die Programmierung mit Go", in: Entwickler Magazin 2.2015

[3] https://golang.org/pkg/net/http/

[4] Stender, Daniel: „Mit Flask Webapplikationen in Python entwickeln", in: Entwickler Magazin 6.2017

[5] https://www.gorillatoolkit.org/pkg/mux

[6] https://golang.org/pkg/html/template/

[7] https://golang.org/pkg/encoding/json/