Ein supereinfacher Webserver mit golang

Michael Dienert

28. Januar 2019

1 Das http.Handler-Interface

Hier ein Auszug aus dem Quellcode des `net/http`-Pakets, das eine API für das Entwickeln von Web-Anwendungen bereitstellt.
Das Interface `http.Handler` ist das Fundament dieser API:

```
package http

type Handler interface {
  ServeHTTP(w ResponseWriter, r *Request)
}

func ListenAndServe(address string, h Handler) error
```

Die `ListenAndServe`-Funktion benötigt also:

1. eine TCP/IP-Server-Adresse, wie z.B. `"localhost:8000"`
2. eine Instanz des `Handler`-Interfaces, an die alle http-Requests zugestellt werden.

1.1 Eine winzigkleine E-Commerce-Anwendung

Der erste Versuch einer Webanwendung könnte so aussehen, wie das Listing auf der Seite 2.
Beschreibung:

type euro float 32 es wird ein eigener Type deklariert

Methode String() die Methode `String()` hat als Empfänger Variablen von Typ `euro` und liefert den Preis in Euro mit 2 Nachkommastellen. Wird nun eine euro-Variable an eine Print-Funktion übergeben, ruft diese die Methode String() auf.

Datenbank unsere Datenbank ist lediglich eine Map: `map[K]V`.

Starten können wir den Server direkt im Verzeichnis mit

```
$ go run shop1.go
```

Beenden einfach mit Ctrl-C.

Die Anwendung listet immer das gesamte Shop-Inventar, egal welchen Pfad wir an die URL anhängen.

```
package main

import (
"fmt"
"net/http"
"log"
)

func main() {
  db := database{"schuhe": 50, "socken": 5, "hemden": 40, "hosen":
      60}
  log.Fatal(http.ListenAndServe("localhost:8000", db))
}

type euro float32

func (e euro) String() string {
  return fmt.Sprintf("EUR %.2f", e)
}  //methode String(), empfaenger euro, rueckabetyp string

// map[K]V string=key, euro=value
type database map[string]euro

func (db database) ServeHTTP(w http.ResponseWriter, r *http.Request)
    {
  for item, price := range db {
    fmt.Fprintf(w, "%s: %s\n", item, price)
  }
}
```

1.2 Webanwendung mit Auswertung des URL-Pfads

Die Handler-Funktion wird erweitert, so dass der absolute Pfad einer URL ausgewertet wird:

```
func (db database) ServeHTTP(w http.ResponseWriter, r *http.Request)
    {
  switch r.URL.Path {
    case "/list":
    for item, price := range db {
      fmt.Fprintf(w, "%s: %s\n", item, price)
    }
    case "/price":
    item := r.URL.Query().Get("item")
    price, ok := db[item]
    if !ok {
      w.WriteHeader(http.StatusNotFound) // 404
      fmt.Fprintf(w, "Ware nicht gefunden: %q\n", item)
      return
    }
    fmt.Fprintf(w, "%s kosten %s\n", item, price)
    default:
    w.WriteHeader(http.StatusNotFound) // 404
    fmt.Fprintf(w, "Seite nicht gefunden: %s\n", r.URL)
  }
}
```

Wie muss die URL aussehen, damit auf der Webseite

```
hosen kosten EUR 60.00
```

angezeigt wird?

1.3 Webanwendung mit Multiplexer (Mux)

Bei sehr vielen verschiedenen Pfaden wäre es bequemer, die Logik für jeden einzelnen Fall in einer separaten Funktion unterzubringen, anstatt eine endlose switch-case-Kette zu bauen.

Für diesen Fall stellt das Paket `net/http` den *Request Multiplexer* `ServeMux` zur Verfügung. Mit diesem wird die Zuordnung zwischen Handler und URL vereinfacht.

Der Multiplexer fasst eine Sammlung einzelner Handler zu einem einzelnen Handler zusammen.

Hier die geänderten Methoden:

```
func main() {
  db := database{"schuhe": 50, "socken": 5, "hemden": 40, "hosen":
      60}
  mux := http.NewServeMux()
  mux.Handle("/list", http.HandlerFunc(db.list))
  mux.Handle("/price", http.HandlerFunc(db.price))
  log.Fatal(http.ListenAndServe("localhost:8000", mux))
}
```

```
http.ResponseWriter, r *http.Request) { for item, price := range db {
    fmt.Fprintf(w, "%s: %s\n", item, price)
  }
}
```

```
func (db database) price(w http.ResponseWriter, r *http.Request) {
  item := r.URL.Query().Get("item")
  price, ok := db[item]
  if !ok {
    w.WriteHeader(http.StatusNotFound) // 404
    fmt.Fprintf(w, "Ware nicht gefunden: %q\n", item)
    return
  }
  fmt.Fprintf(w, "%s kosten %s\n", item, price)
}
```

1.4 Besonderheiten des Go-Interface-Mechanismus

Das Registrieren der Handler enthält ein paar Besonderheiten von go:

Z.B. die Zeile `mux.Handle("/list", http.HandlerFunc(db.list))`

`db.list` (ja, ohne runde Klammern!) ist ein *Methoden-Wert*, d.h. ein Wert vom Typ

```
func(w http.ResponseWriter, req *http.Request)
```

Da diese Funktion aber keine Methoden hat, implementiert sie auch nicht das http.Handler-Interface und kann nicht direkt an `mux.Handle` übergeben werden. [1]

Der Ausdruck http.HandlerFunc(db.list) ist kein Funktionsaufruf (!) sondern eine Type-Konvertierung, da http.HandlerFunc ein type ist, der folgendermassen definiert ist:

```
package http

type HandlerFunc func(w ResponseWriter, r *Request)

func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request){
  f(w,r)
}
```

`HandlerFunc` ist nun ein Function-Type der Methoden hat und das Interface `http.Handler` implementiert.

ServeHTTP macht aber nichts anderes, als die darunterliegende Funktion `f(w,r)` aufzurufen.

`HandlerFunc` wird damit zu einem Adapter, der es einer Funktion ermöglicht ein Interface zu implementieren, bei der die Funktion und die einzelne Methode des Interfaces die *gleiche Signatur* haben.

Wozu das Ganze? Mit diesem Trick wird es möglich, dass ein einzelner Type wie z.B. `database` das `http.Handler`-Interface auf verschiedenen Wegen implementiert: einmal mit seiner `list`-Methode, einmal mit seiner `price`-Methode usw.

1.5 Noch einfacher

Damit das Ganze noch einfacher wird, gibt es eine Komfort-Funktion, die die in 1.4 beschriebene Konversion für uns gleich mit erledigt:

```
mux.HandleFunc("/list", db.list)
mux.HandleFunc("/price", db.price)
```

1.6 Am einfachsten

`net/http` bringt ohnehin schon einen globalen Multiplexer mit: (`DefaultServeMux`). Ausserdem gibt es die Package-Level-Funktion `http.HandleFunc`.

Wenn wir also nicht mehrere verschiedene Multiplexer benötigen (das dürfte der Standardfall sein: ein http-Server an Port 80 **oder** 8080 aber nicht beide), können wir den immer vorhandenen Default-Multiplexer nehmen.

Das passiert immer automatisch dann, wenn wir an `http.ListenAndServe` als Multiplexer einen Null-Pointer (nil) übergeben:

```
func main(){
  db := database{ ... }
  http.HandleFunc("/list", db.list)
  http.HandleFunc("/price", db.price)
  log.Fatal(http.ListenAndServe("localhost:8000", list))
}
```

[1] in der englischen Literatur heist es bei go: a method *satisfies* an interface

2 Aufgaben

2.1 Codebeispiele testen

Teste alle oben beschriebenen Code-Beispiele. (http1.go - http4.go)

2.2 Zusätzliche Handler

Füge zusätzliche Handler hinzu, so dass Anwender *create, read, update* und *delete* - Aktionen auf die Datenbank ausführen können.
Zum Beispiel durch Eingabe folgender URL:

```
/update?item=socken&price=6
```

Fehlerbehandlung nicht vergessen.

2.3 Templates

Der Handler von `/list` soll seine Ausgabe als HTML-Tabelle erzeugen (kein plain-text). Dazu gibt es das `html/template`-Package