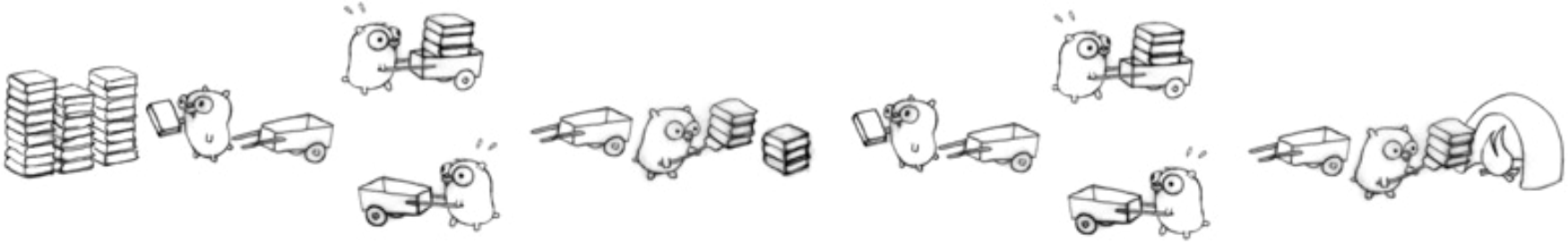
Web-Services mit Go

Go-Features an Hand von Beispielen

Sebastian 'tokkee' Harl
<sh@tokkee.org>



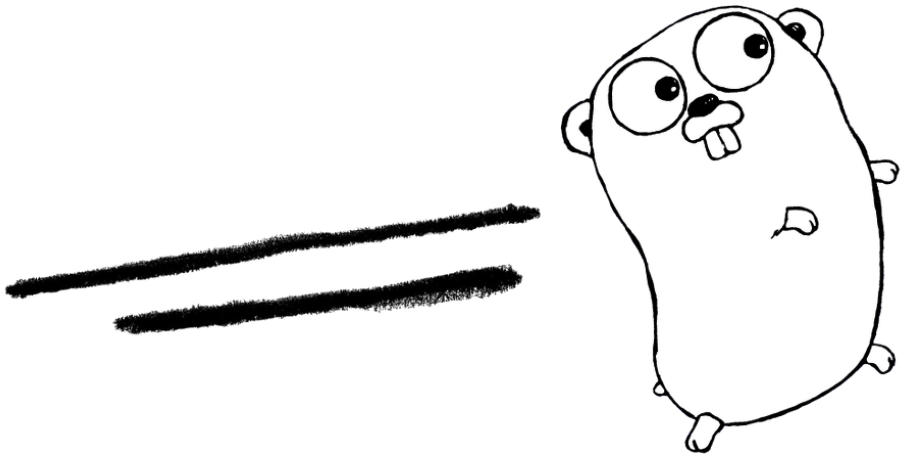
Grazer LINUXTAGE
29. + 30. April 2016 www.linuxtage.at

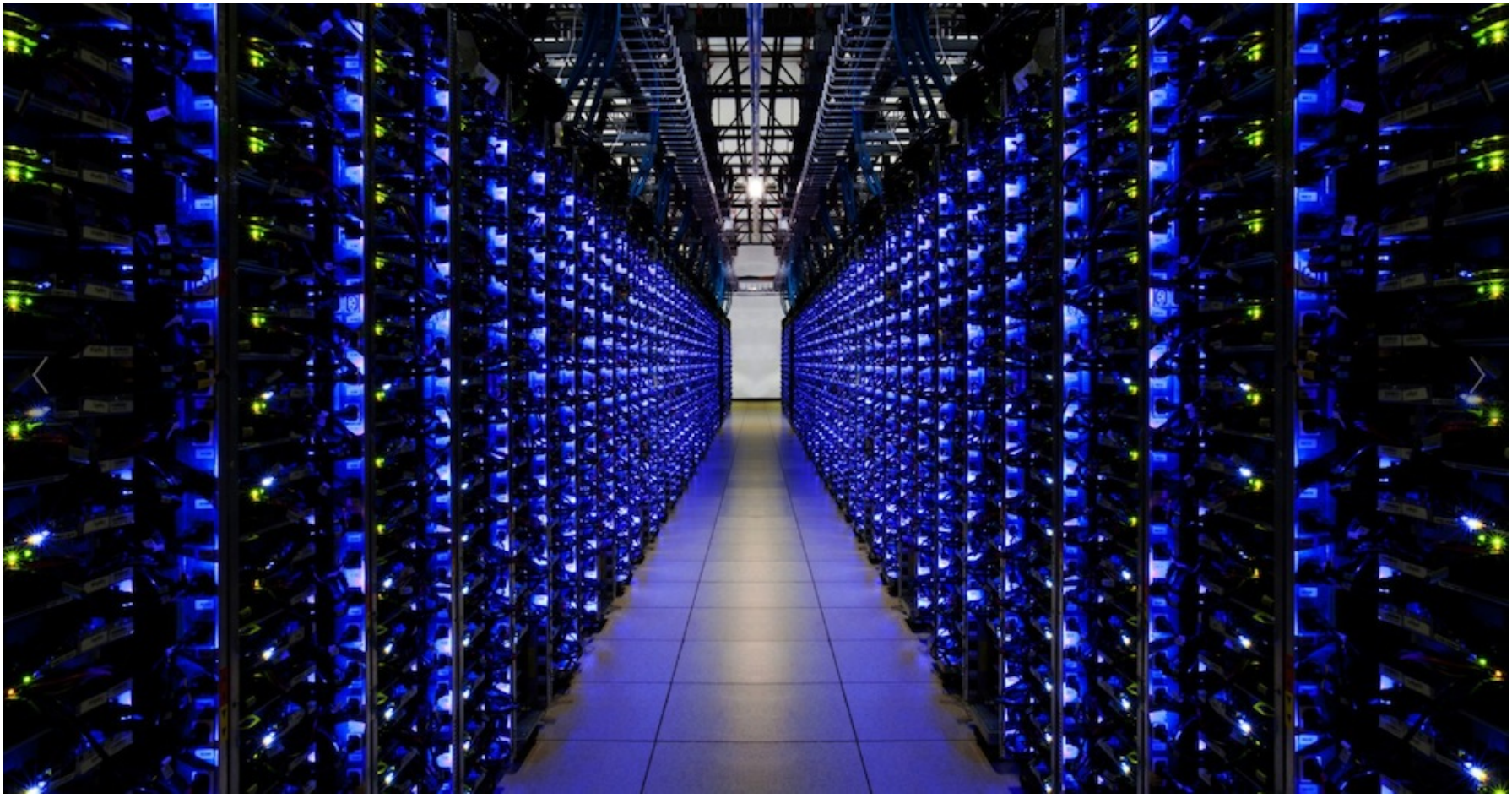
Überblick

Was ist Go?

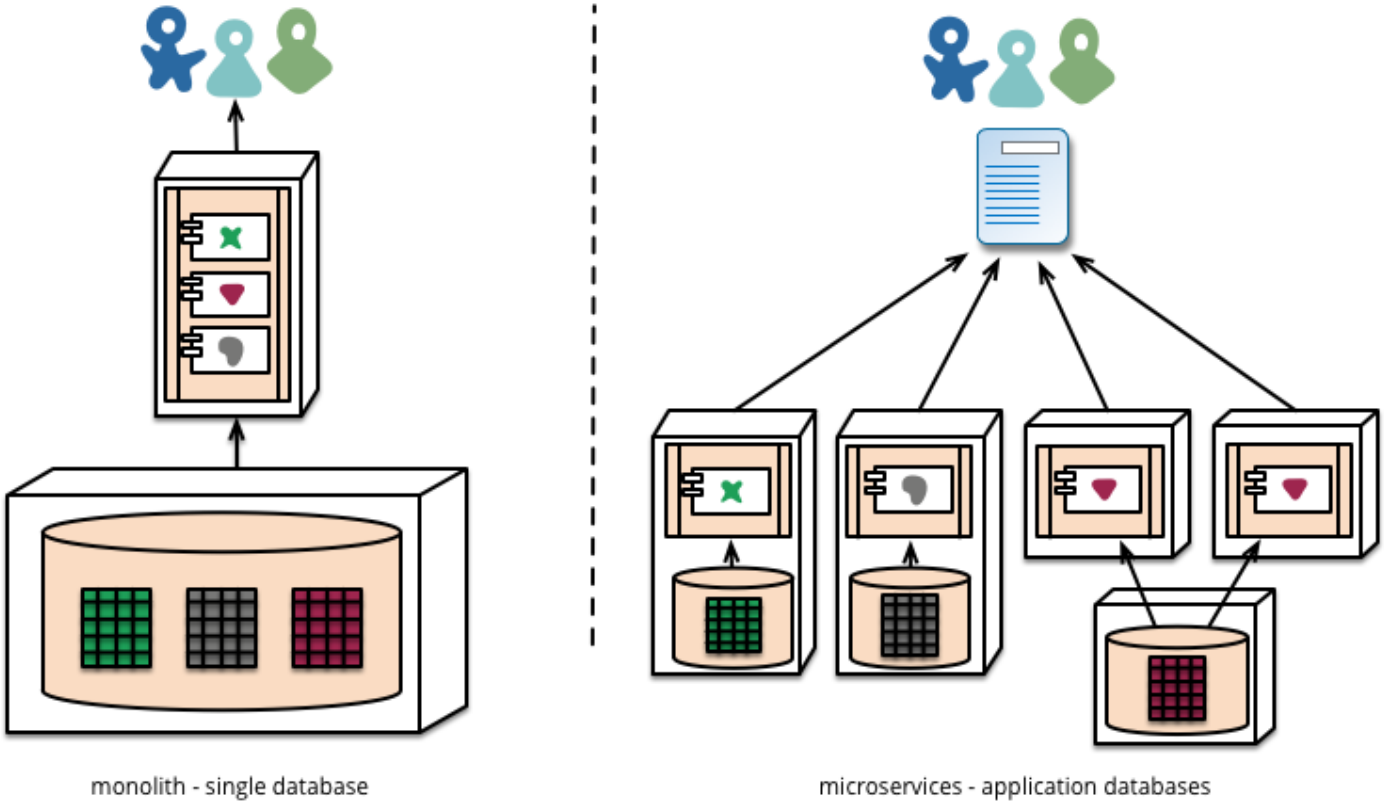
Go is an open source programming language that makes it easy to build simple, reliable, and efficient software.

`https://golang.org`





Web-Services



Quelle: `http://martinfowler.com/articles/microservices.html`

Web-Services

- HTTP Frontend
  - Viele parallele Client-Anfragen
  - Eine oder mehrere Verbindungen zu Backends
- Backend („Business Logic“)
  - Viele parallele Anfragen vom Frontend
  - Eine oder mehrere Datenbank-Verbindungen oder Interaktion mit anderen Backends
- Datenbank

$\Rightarrow$ skalierbare Micro-Services / **lose Kopplung**

Auch: Gleiche Prinzipien bei Integration mit anderen Lösungen

Einschub: Go Standard Bibliothek

`https://golang.org/pkg/`

- Crypto
- Datenbanken
- Go Parser
- Netzwerk, HTTP, SMTP, etc.
- Datenstrukturen

Mehr? ⇒ `https://godoc.org/`

Ein Webserver

```
1  import (...)
2
3  func main() {
4    http.HandleFunc("/hallo", sageHallo)
5    log.Fatal(http.ListenAndServe(":9999", nil))
6  }
7
8  func sageHallo(w http.ResponseWriter, r *http.Request) {
9    fmt.Fprintf(w, "Hallo %s", r.RemoteAddr)
10 }
```

- https://golang.org/pkg/log/
- https://golang.org/pkg/net/http/

Ein Webserver – Templates

```
1  var tmpl = template.Must(
2    template.New("results").Parse(`
3  <html><head>
4  <title>{{.Title}}</title>
5  </head>
6
7  <body>
8  <h1>Hallo {{.Name}}</h1>
9  </body></html>
10 `))
```

- https://golang.org/pkg/html/template/

Ein Webserver – Templates (2)

```
1  func sageHallo(...) {
2    d := struct {
3      Title, Name string
4    }{"Hallo Welt", r.RemoteAddr}
5
6    var buf bytes.Buffer
7    if err := tmpl.Execute(&buf, d); err != nil {
8      http.Error(w, err.Error(), http.StatusInternalServerError)
9      return
10   }
11   io.Copy(w, &buf)
12 }
```

- https://golang.org/pkg/bytes/
- https://golang.org/pkg/io/

Einschub: io.Writer

- Warum funktioniert fmt.Printf, tmpl.Execute, http.Error, io.Copy eigentlich mit dem http.ResponseWriter und bytes.Buffer?

```
1 package io
2 type Writer interface {
3     Write(p []byte) (n int, err error)
4 }
```

⇒ **Sehr** einfaches Interface

⇒ `http.ResponseWriter` und `bytes.Buffer` implementieren es

Viele andere Beispiele ...

Viele Backend-Abfragen

```
1  func Query(*Request) (*Response, error) { ... }
2
3  func anfrage(w http.ResponseWriter, r *http.Request) {
4    requests := []*Request{...}
5
6    responses := make([]*Reponse, len(requests))
7    errCh := make(chan error, len(requests))
8
9    for i, req := range requests {
10     go func(req *Request) {
11       var err error
12       responses[i], err = Query(req)
13       errCh <- err
14     }(req)
15   }
16
17   // ...
```

Viele Backend-Abfragen (2)

```
1   ...
2   timeout := time.After(50*time.Millisecond)
3
4   for range requests {
5     select {
6     case err := <-errCh:
7       if err != nil {
8         http.Error(w, err.Error(), http.StatusBadRequest)
9         return
10      }
11    case <-timeout:
12      http.Error(w, "timeout", http.StatusRequestTimeout)
13      return
14    }
15  }
16  // Alle Ergebnisse verfügbar.
```

- Siehe auch `https://golang.org/pkg/sync/#WaitGroup`

Spezielle Fehlerbehandlung

```
1   go func(req *Request) {
2     var err error
3     defer func() {
4       if e := recover(); e != nil {
5         err = fmt.Errorf("error while querying backend: %v", e)
6       }
7       errCh <- err
8     }()
9
10    responses[i], err = Query(req)
11  }(req)
```

Backends – gRPC

Beispiel: Backend Kommunikation / API

A high performance, open source, general RPC framework that puts mobile and HTTP/2 first.

`https://grpc.io`

gRPC Überblick

`https://github.com/google/protobuf`

- gRPC basiert auf Protocol Buffers
- Unterstützung diverser Sprachen
  C++, Java, Go, Python, ...

gRPC Beispiel

```
1  syntax "proto3";
2
3  package mein_service;
4
5  service Backend {
6    rpc Query(QueryRequest) return (QueryResponse);
7
8    // ...
9  }
10
11 message QueryRequest {
12   string query = 1;
13 }
14
15 message QueryResponse {
16   string type = 1;
17   int64 n = 2;
18 }
```

gRPC mit Go

- Die „protobuf“ Datei muss mittels des Protobuf Compilers und einer gRPC Compiler-Erweiterung übersetzt werden
- Der Compiler erzeugt Go Code, welcher Interfaces und generischen Code erzeugt
- Das Interface entspricht im Wesentlichen der `service` Definition
- Das Interface muss für den Server implementiert werden
- Generischer Client-Code sollte ausreichen
  $\rightarrow$ API Entwurf!

`https://github.com/grpc/grpc-go`
`https://golang.org/x/net/context`

gRPC mit Go: Server

```
1  import pb "tokkee.net/mein_service/service_proto"
2
3  type server struct{}
4
5  func (*server) Query(ctx context.Context,
6    in *pb.QueryRequest) (*pb.QueryResponse, error) {
7
8    n, err := runQuery(in.Query)
9    if err != nil {
10     return nil, err
11   }
12   return &pb.QueryResponse{
13     Type: "irgendwas",
14     N:    n,
15   }, nil
16 }
```

gRPC mit Go: Server (2)

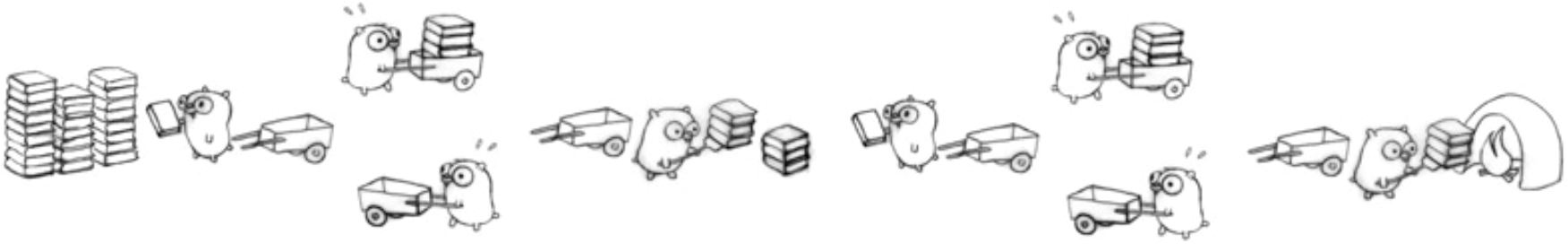
```
1  func main() {
2    l, err := net.Listen("tcp", port)
3    if err != nil {
4      log.Fatal(err)
5    }
6
7    s := grpc.NewServer()
8    pb.RegisterBackendServer(s, &server{})
9    s.Serve(l)
10 }
```

gRPC mit Go: Client

```
1  func main() {
2    ctx := context.Background()
3
4    conn, err := grpc.Dial("localhost:50051", grpc.WithInsecure())
5    if err != nil {
6      log.Fatal(err)
7    }
8    defer conn.Close()
9    c := pb.NewBackendClient(conn)
10
11   res, err := c.Query(ctx, &pb.QueryRequest{
12     Query: "ein query",
13   })
14   if err != nil {
15     log.Fatal(err)
16   }
17   fmt.Printf("Antwort vom Typ %q: %d\n", res.Type, res.N)
18 }
```

Danke für die Aufmerksamkeit

Fragen, Kommentare?



`https://golang.org` — `https://github.com/grpc/grpc-go`