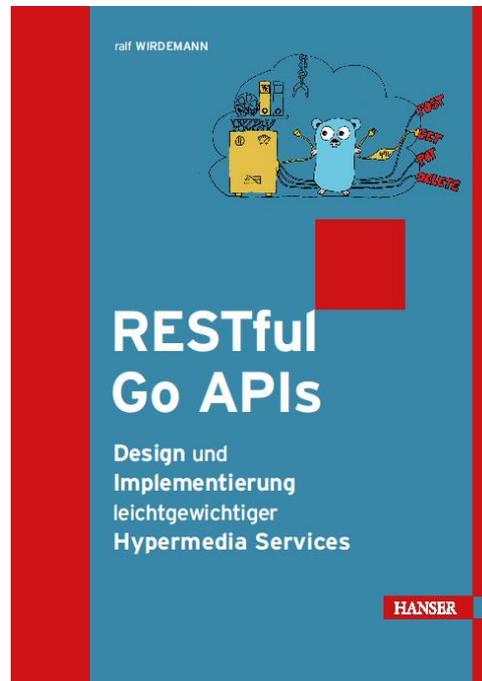


HANSER



Leseprobe

zu

RESTful Go APIs von Ralf Wirdemann

ISBN (Buch): 978-3-446-45709-6

ISBN (E-Book): 978-3-446-45978-6

Weitere Informationen und Bestellungen unter

<https://www.hanser-fachbuch.de/buch/RESTful+Go+APIs/9783446460430>

sowie im Buchhandel

© Carl Hanser Verlag, München

Inhalt

1	Einleitung	1
1.1	Alle bauen APIs.....	1
1.2	Web-Frameworks.....	2
1.3	Alle wollen Go.....	2
1.4	Warum REST?.....	3
1.5	Aufbau des Buches.....	4
1.6	Disclaimer.....	4
1.7	Gebrauchsanleitung.....	5
1.8	Code-Beispiele.....	5
2	Go	7
2.1	Installation.....	7
2.2	Der Go-Workspace.....	8
2.3	Test der Installation.....	9
2.4	Programmstruktur.....	9
2.5	Packages.....	10
2.6	Funktionen.....	11
2.7	Variablen.....	11
2.8	Typen.....	12
2.9	Schleifen.....	14
2.10	Verzweigungen.....	15
2.11	Methoden.....	17
2.12	Pointer.....	18
2.13	Interfaces.....	19
2.14	Arrays und Slices.....	21
	2.14.1 Arrays.....	21
	2.14.2 Slices.....	21
	2.14.3 Polymorphe Arrays.....	23

2.15	Maps	23
2.16	Verzögerungen, Panic und Recover	24
2.17	Was sonst noch?	26
3	REST	27
3.1	Überblick	27
3.2	Ressourcen	29
3.3	Uniform Resource Identifier	29
3.4	HTTP-Methoden	30
3.4.1	GET	30
3.4.2	POST	31
3.4.3	PUT	31
3.4.4	DELETE	31
3.4.5	PATCH	31
3.4.6	HEAD	31
3.4.7	OPTIONS	31
3.5	Repräsentationen	32
3.6	Statuscodes	32
3.6.1	2xx Erfolg	33
3.6.2	3xx Umleitung	33
3.6.3	4xx Client-Fehler	33
3.6.4	5xx Server-Fehler	33
3.7	Hypermedia	34
3.8	REST als Architekturstil	35
3.8.1	Client-Server	35
3.8.2	Zustandslosigkeit	35
3.8.3	Cacheability	36
3.8.4	Uniform Interface	36
3.8.5	Layered System	36
3.8.6	Code on Demand	37
3.9	Zusammenfassung	37
4	HTTP und JSON	39
4.1	Das Package net/http	39
4.1.1	Request Handling	39
4.1.2	Level-2-Services	43
4.2	Das Package encoding/json	45
4.2.1	JSON-Encoding	46
4.2.2	JSON-Decoding	48
4.3	HTTP und JSON	49
4.4	Abschluss	50

5	Restvoice	51
5.1	Vision	51
5.2	Use Cases	52
5.3	Domainmodell	53
5.4	Ressourcen.....	55
5.5	Kern-Use-Cases	55
5.5.1	Rechnung erstellen	57
5.5.2	Buchung zufügen	60
5.5.3	Buchung löschen	63
5.5.4	Rechnung abschließen	64
5.5.5	Rechnung zustellen	65
5.6	Kritik am Entwurf	66
5.6.1	Sicherheit.....	66
5.6.2	Testbarkeit	67
5.6.3	Kopplung	67
5.6.4	RESTfulness	68
5.7	Warum Go?	68
5.8	Abschluss	69
6	Design	71
6.1	Layering	72
6.2	Domain-driven Design.....	72
6.3	Hexagonale Architektur	73
6.3.1	Beispiel	75
6.3.2	Der Use Case ELIZA.....	75
6.3.3	Ports und Adapter	76
6.4	Use Case „Rechnung erstellen“	77
6.4.1	HTTP ⇒ Invoice	78
6.4.2	Geschäftslogik	79
6.4.3	Invoice ⇒ HTTP	80
6.4.4	Bootstrapping.....	80
6.5	Use Case „Tätigkeit buchen“	82
6.6	Use Case „Rechnung abschließen“	83
6.7	Use Case „Rechnung anfordern“	85
6.8	Was haben wir erreicht?.....	86
6.8.1	Entkopplung	87
6.8.2	Wartbarkeit.....	87
6.8.3	Testbarkeit	88

7	Testing	89
7.1	Unit Tests	89
7.2	Unit Tests in Go	90
7.3	HTTP-Tests in Go	92
7.4	Was testen?	93
7.5	Beispiel: Rechnung abschließen	94
7.5.1	Testszenario	95
7.5.2	Method under Test	95
7.5.3	Unit Test	97
7.5.4	Integrationstest	97
7.6	Fake- und Mock-Objekte	99
7.6.1	Fake-Objekte	99
7.6.2	Mock-Objekte	101
7.7	Unit Test mit Fake-Objekt	102
7.8	HTTP-Test	102
7.9	API-Test	103
7.10	Testseparation	105
7.10.1	Build Tags: -tags	105
7.10.2	Short Modus: -short	106
7.10.3	Einzelne Tests ausführen: -run	106
7.11	Test Coverage	106
7.12	Abschluss	108
8	Hypermedia als Motor	109
8.1	HATEOAS	110
8.2	Hypermedia Rechnungsstellung	110
8.2.1	Zustandsübergänge	111
8.2.2	Wie sage ich es meinem Client?	112
8.3	Hypertext Application Language	113
8.4	HAL in Go	116
8.4.1	Geschäftslogik	116
8.4.2	Media Type „application/hal+json“	117
8.5	Hypermedia-Clients	119
8.6	Resource Expansion	121
8.7	CRUD vs. Hypermedia	124
8.8	Abschluss	125

9	Sicherheit	127
9.1	Client-Authentifizierung.....	127
9.1.1	HTTP Basic.....	128
9.1.2	HTTP Digest.....	132
9.1.3	JSON Web Token.....	134
9.2	Server-Authentifizierung.....	139
9.3	Autorisierung.....	141
9.4	Verschlüsselung.....	143
9.5	Abschluss.....	144
10	Skalierbarkeit	147
10.1	Horizontale Skalierung.....	147
10.2	Zustandslosigkeit.....	148
10.2.1	Sitzungsstatus.....	148
10.2.2	Clientstatus.....	148
10.2.3	Ressourcenstatus.....	149
10.3	Restvoice skalieren.....	150
10.4	Vertikale Skalierung.....	150
10.4.1	Go-Routinen.....	151
10.4.2	Restvoice vertikal skalieren.....	151
10.5	Abschluss.....	153
11	Caching	155
11.1	Arten von Caches.....	155
11.1.1	Lokaler Cache.....	155
11.1.2	Proxy Cache.....	156
11.1.3	Reverse Proxy Cache.....	156
11.2	HTTP Caching.....	156
11.2.1	GET.....	156
11.2.2	Cachebare Responses.....	156
11.2.3	Cache-Validierung.....	157
11.2.4	Cache-Invalidierung.....	159
11.3	Restvoice Caching.....	159
11.4	Activity Caching.....	159
11.4.1	Cacheable Responses.....	159
11.4.2	Cache-Validierung.....	161
11.4.3	Invalidierung.....	163
11.4.4	Alternative ETags.....	164
11.5	Abschluss.....	164

12 Wie geht es weiter?	165
12.1 Danke!	165
12.2 Feedback	166
Literatur	167
Stichwortverzeichnis	169

2

Go

Go ist eine schlanke Programmiersprache und kommt mit 25 Keywords aus. Go ist weder ausdrücklich funktional noch ausdrücklich objektorientiert. Dennoch lassen sich beide Paradigmen in Go umsetzen. Dieses Kapitel führt in die Grundlagen der Programmiersprache Go ein.

Das Kapitel beginnt mit der Installation einer Go-Entwicklungsumgebung und der Beschreibung des Go-Workspace. Anschließend werden die Basiskonstrukte von Go beschrieben: Packages strukturieren Go-Dateien auf oberster Ebene und definieren die Sichtbarkeit enthaltener Elemente. Funktionen implementieren Programmlogik mithilfe von Schleifen, Bedingungen und Variablen. Typen stehen entweder in Form von Standard- oder benutzerdefinierten Typen zur Verfügung.

Nach Einführung dieser grundlegenden und aus anderen Programmiersprachen bekannten Sprachkonstrukte fährt das Kapitel mit den Besonderheiten von Go fort: Pointer, Interfaces und Methoden. Pointer ermöglichen die explizite Unterscheidung von Werte- und Referenztypen. Interfaces trennen Schnittstelle und Implementierung von Typen. Methoden implementieren Funktionen auf benutzerdefinierten Typen, die auf Instanzen des jeweiligen Typs gebunden werden.

Im letzten Abschnitt werden mit Arrays und Maps zwei der neben den Standardtypen wichtigsten Go-Typen beschrieben. Arrays implementieren typisierte Listen fester Länge. Maps dienen der Verwaltung typisierter Key-/Value-Paare.

■ 2.1 Installation

Go-Distributionen stehen zum Binär-Download für MacOS, Windows und Linux unter <https://golang.org/dl/> bereit. Unter MacOS wird die Go-Distribution nach `/usr/local/go` installiert. Damit die Go-Tools im Terminal funktionieren, muss die PATH-Variable um den Pfad `/usr/local/go/bin` erweitert werden. Führen Sie das Go-Tool in einem neuen Terminalfenster mit dem Parameter `version` aus:

```
$ go version
go version go1.11 darwin/amd64
```

Entspricht die Ausgabe dem Beispiel, dann hat alles funktioniert, und die Go-Entwicklungs-umgebung steht bereit. Weitere Installationshinweise sowie Anleitungen für die Go-Installation unter Windows und Linux finden Sie hier <https://golang.org/doc/install>.

■ 2.2 Der Go-Workspace

Das grundlegende Konzept einer Go-Entwicklungsumgebung ist der *Go-Workspace*. Der Go-Workspace ist ein zentrales Verzeichnis, das über die Umgebungsvariable *GOPATH* referenziert wird. Unterhalb des *GOPATH* finden sich die drei Unterverzeichnisse *bin*, *pkg* und *src*:

```
bin/
  restvoice
pkg/
  linux_amd64/
    github.com/rwirdemann/restvoice/
      usecase.a
src/
  github.com/rwirdemann/restvoice/
    main.go
    usecase/
      create_invoice.go
```

Go-Programme werden compiliert und zu einem statischen Binary gelinkt. Die resultierende Binärdatei landet im Verzeichnis *bin* des Go-Workspace. Zusammengehörige Dateien, sogenannte *Packages*, werden zu einer *package.a*-Datei übersetzt und in eines der plattformspezifischen Unterverzeichnisse von *pkg* abgelegt. Das Unterverzeichnis *src* enthält die Dateien mit den Go-Quellen, geordnet in Unterverzeichnisse, die die Package-Struktur widerspiegeln.

Die zugrunde liegende Idee des Go-Workspace ist, dass alle Projekte inklusive der benötigten Abhängigkeiten an zentraler Stelle abgelegt werden. Projekte werden schnell gefunden und können sich benötigte Bibliotheken teilen. Die Nutzung mehrerer Workspaces ist durch die Verwendung der Umgebungsvariable *GOPATH* einfach möglich.

Der Go-Workspace ist prinzipiell eine gute Idee, birgt aber auch eine Reihe praktischer Probleme. So ist es nicht ohne Weiteres möglich, unterschiedliche Versionen eines Packages im selben Workspace zu verwalten. Es ist auch nicht möglich, eine Bibliothek zu veröffentlichen, die eine bestimmte Version einer abhängigen Bibliothek benötigt.

Mit Version 1.5 wurde Go um eine Dependency-Management-Technik, das sogenannte *Vendoring* erweitert. Mit *Vendoring* kann man einem Projekt weitere Abhängigkeiten aus einem projektlokalen *vendor*-Verzeichnis zufügen. Seit Go-Version 1.11 steht mit *Go Modules* eine Ablösung des *Vendoring* ins Haus. Die Beispiele in diesem Buch funktionieren *GOPATH*-basiert. Die Beschäftigung mit *Vendoring* oder besser *Go Modules* ist an dieser Stelle nicht erforderlich. Spätestens wenn es in die Praxis geht, müssen Sie sich mit dem Thema *Dependency Management* beschäftigen. Das Go-Wiki enthält eine gute Einführung zur Funktionsweise und Benutzung von *Go Modules* [The18].

■ 2.3 Test der Installation

Erstellen Sie den Go-Workspace mit den drei Unterverzeichnissen *bin*, *pkg* und *src* und exportieren Sie dessen Verzeichnis in der Umgebungsvariable *GOPATH*:

```
$ cd $HOME
$ mkdir -p go/bin go/pkg go/src
$ export GOPATH=$HOME/go
```

Anschließend wird im Verzeichnis *src* eine Datei *main.go* mit folgendem Inhalt erstellt:

```
1 package main
2
3 func main() {
4     println("Hello, Jo")
5 }
```

Die Funktion *main* im Package *main* ist der Einstiegspunkt in ein Go-Programm und wird beim Start des Programms ausgeführt. Das Kommando *go run* kompiliert und führt das Programm aus:

```
$ cd $GOPATH/src
$ go run main
Hello, Jo
```

Das Kommandozeilentool *go* ist das zentrale Werkzeug einer Go-Distribution. Entsprechend parametrisiert lassen sich damit alle wichtigen Entwicklungsaufgaben ausführen. Einige Beispiele:

```
go build main.go # Übersetzt und legt das Binary im aktuellen Verzeichnis ab
go install main.go # Übersetzt und legt das Binary im Verzeichnis "bin" ab
go test # Führt die Tests im aktuellen Package aus
```

Ein vollständige Dokumentation der Go-Tools finden Sie unter <https://golang.org/cmd/go>.

■ 2.4 Programmstruktur

Go-Quellcode ist auf Dateien mit dem Suffix *.go* verteilt. Die Struktur einer Go-Datei wird auf oberster Ebene durch sechs Keywords bestimmt:

```
package
import
var
const
type
func
```

Das erste Statement einer Go-Datei ist immer *package*, gefolgt von einem oder mehreren *import*-Statements zum Import referenzierter Packages. Anschließend folgen Variablen- und Konstantendeklarationen sowie Typ- und Funktionsdefinitionen in beliebiger Reihenfolge und Anzahl.

■ 2.5 Packages

Go-Files werden in Packages organisiert. Alle Dateien eines Verzeichnisses gehören zum selben Package. Der letzte Teil des Pfadnamens bestimmt den Package-Namen. Gemäß dieser Konvention liegt die Datei *memory.go* im Verzeichnis *\$GOPATH/go101/cache* und gehört zum Package *cache*:

```

1 package cache
2
3 func Write(key string, value string) {
4     ...
5 }
```

Packages definieren Namensräume. Die Sichtbarkeit von Typen, Funktionen, Variablen und Konstanten wird mithilfe von Groß- und Kleinschreibung definiert. Alles Kleingeschriebene ist ausschließlich innerhalb des Packages sichtbar. Alles Großgeschriebene ist auch außerhalb des Packages sichtbar. Packages werden über das *import*-Statement importiert. Öffentliche Typen, Variablen und Funktionen werden genutzt, indem der Package-Name dem importierten Element vorangestellt wird:

```

1 package main
2
3 import "cache"
4
5 func main() {
6     cache.Write("1", "Kalle")
7 }
```

Die Funktion *main* importiert das Package *cache* und ruft die exportierte Funktion *cache.Write* auf. Das *main*-Beispiel zeigt eine Ausnahme in Bezug auf die Namenskonvention für Packages: Die Go-Datei mit der Funktion *main* gehört zum Package *main*, unabhängig vom Verzeichnis, in dem sie liegt.

Initialisierung

Jede Datei eines Packages kann eine *init*-Funktion enthalten, in der paketweite Initialisierungsaufgaben ausgeführt werden. Init-Funktionen werden beim Laden eines Packages automatisch ausgeführt:

```

1 package auth
2
3 var kf []byte
4
5 func init() {
6     if kf, err := ioutil.ReadFile("public.key"); err != nil {
7         log.Fatalf("Could not open public key file: %s", "public.key")
8     }
9 }
```

Init-Funktionen sind Package-intern und können aus keinem anderen Package heraus aufgerufen werden. Ein anonymer Package-Import sorgt für die Ausführung von *init*, ohne dass eine öffentliche Funktion des importierten Packages aufgerufen werden muss. Dies ist zum Beispiel dann sinnvoll, wenn eine Bibliothek Initialisierungscode enthält, der nie explizit aufgerufen wird, für das Funktionieren des Programmes aber einmalig ausgeführt werden muss. Das Laden eines Datenbanktreibers ist ein Beispiel:

```
import _ "github.com/go-sql-driver/mysql"
```

Der Import lädt ausschließlich den MySQL-Datenbanktreiber. Anschließend werden die SQL-Funktionen des Go-Standard-Package *database/sql* genutzt.

■ 2.6 Funktionen

Funktionen in Go werden mit dem Schlüsselwort *func*, gefolgt vom Namen, einer optionalen Parameterliste, keinem, einem oder mehreren Rückgabewerten deklariert:

```
1 func inc(i int) int {
2     return i + 1
3 }
```

Funktionen mit mehreren Rückgabewerten müssen deren Typen in Klammern angeben:

```
1 func swap(x int, y int) (int, int) {
2     return y, x
3 }
```

Return-Werte können Namen haben. Benannte Return-Werte können ohne explizite Deklaration im Funktions-Body verwendet werden. Ein parameterloses Return-Statement liefert den zuletzt an diese Variable zugewiesenen Wert zurück:

```
1 func inc(i int) (j int) {
2     j = i + 1
3     return // Liefert "j"
4 }
```

■ 2.7 Variablen

Variablen werden mit dem Schlüsselwort *var* oder per Direktzuweisung deklariert:

```
1 var firstname string
2
3 func name() string{
4     lastname := "Brunner"
5     return firstname + " " + lastname
6 }
```

Bei der Direktzuweisung erkennt der Compiler den Typ der Variablen, sodass die explizite Angabe des Typs *string* entfallen kann und die Variable direkt initialisiert wird. Eine Direktzuweisung funktioniert nur innerhalb von Funktionen. Außerhalb von Funktionen deklarierte Variablen müssen mit *var* deklariert werden. Wird eine per *var* deklarierte Variable direkt initialisiert, kann die Typangabe entfallen:

```
var firstname = "Jo"
```

Die Sichtbarkeit einer Variablen wird durch den sie umgebenden Block bestimmt. Außerhalb von Funktionen deklarierte Variablen sind in allen Funktionen des Packages sichtbar. Ist die Variable groß geschrieben, dann ist sie außerhalb ihres Packages sichtbar und wird durch den vorangestellten Package-Namen adressiert. Im folgenden Beispiel exportiert das Package *mysql* die Variable *Db*:

```
1 package mysql
2
3 import (
4     "database/sql"
5 )
6
7 var Db *sql.DB
8
9 func init() {
10     Db, _ = sql.Open("mysql", "user:password@dbname")
11 }
```

Die exportierte Variable *Db* wird im Package *main* durch Voranstellen des Package-Namens *mysql* verwendet:

```
1 package main
2
3 import (
4     "go-basics/mysql"
5 )
6
7 func main() {
8     mysql.Db.Ping()
9 }
```

■ 2.8 Typen

Go ist eine statische, getypte Programmiersprache. Jede Variable, jeder Funktionsparameter und jeder Rückgabewert hat einen Typ, dessen korrekte Verwendung vom Compiler sichergestellt wird. Das Typsystem von Go ist leichtgewichtig. Typinferenz sorgt dafür, dass ein Typ nur dann anzugeben ist, wenn der Compiler ihn nicht aus dem Kontext schließen kann.

Tabelle 2.1 Die Standardtypen von Go

Typ	Nullwert	Hinweis
bool	false	
string	␣	Leerzeichen
int int8 int16 int32 int64	0	
uint, uint8, uint16, uint32 uint64 uintptr	0	
byte	0	Alias für uint8
rune	0	Alias für int32, Unicode Code Point
float32	0.0	
float64	0.0	
complex64	0+0i	
complex128	0+0i	

Go unterscheidet zwischen Standard- und benutzerdefinierten Typen. Tabelle 2.1 gibt einen Überblick über die Go-Standardtypen mit ihren jeweiligen Nullwerten. Der Nullwert eines Typs ist der Wert, mit dem nicht explizit initialisierte Variablen belegt werden.

Aufbauend auf die Go-Standardtypen können benutzerdefinierte Typen definiert werden. Das folgende Beispiel definiert den benutzerdefinierten Typ *Location* basierend auf *string*:

```

1 type Location string
2
3 func foo() {
4     var city Location
5     city = "Hamburg"
6     fmt.Printf("City: %s\n", city)
7 }
```

Zusammengesetzte Typen

Mithilfe des Keyword *struct* werden mehrere Attribute zu einem zusammengesetzten Typen kombiniert:

```

1 type Address struct {
2     Street string
3     City   Location
4 }
```

Zusammengesetzte Typen werden per Literal instanziiert, indem dem Typnamen eine in geschweiften Klammern angegebene Parameterliste übergeben wird:

```
address := Address{Street: "Oxford Street", City: "London"}
```

Die Angabe der Parameternamen ist nur bei partieller Initialisierung erforderlich. Nicht explizit initialisierte Attribute werden mit ihrem Nullwert initialisiert. Bei vollständiger

Initialisierung reicht die Kurzschreibweise:

```
address := Address{"Oxford Street", "London"}
```

Zusammengesetzte Typen können beliebig geschachtelt werden:

```
1 type Contact struct {
2     Name string
3     Mobile string
4     Home Address
5 }
6
7 address := Address{"Oxford Street", "London"}
8 contact := Contact{Name: "Jo", Home: address}
```

Der Zugriff auf die Attribute eines zusammengesetzten Typs erfolgt über die Punktnotation:

```
1 fmt.Println(contact.Name)
2 fmt.Println(contact.Home.Street)
```

Alternativ zur Angabe eines expliziten Attributnamens lassen sich zusammengesetzte Typen auch anonym einbetten. Ein anonym eingebettetes Struct besteht nur aus seinem Typ:

```
1 type Contact struct {
2     Name string
3     Mobile string
4     Address
5 }
```

Die Felder eingebetteter Structs werden direkt referenziert:

```
fmt.Println(contact.Street)
```

Eingebettete Structs werden per Literal oder Zuweisung initialisiert:

```
1 contact := Contact{Address: Address{"Oxford Street", "London"}}
2 contact.Street = "Oxford Street"
```

■ 2.9 Schleifen

Go kennt nur eine Schleife, die for-Schleife. Je nach Parameter erfüllt die Schleife unterschiedliche Zwecke:

```
for i := 0; i < 10; i++ // Index-basierte Iteration
for i < 10             // while
for                   // for ever
for i, v := range foo // Index- und Werte-basierte Iteration
```

Die `for`-Bedingung ist immer ohne Klammern, der `for`-Body immer in geschweiften Klammern anzugeben:

```
1 i := 0
2 for i < 10 {
3     fmt.Println(i)
4     i++
5 }
```

In Kombination mit `range` iteriert `for` über eine Liste von Werten und liefert für jeden Schleifendurchlauf den jeweiligen Listenwert sowie dessen Listenindex:

```
1 for i, v := range []string{"a", "b", "c"} {
2     fmt.Println("Index:", i, "Wert:", v)
3 }
```

Wird die Angabe der Werte-Variablen weggelassen, liefert `range` nur den jeweiligen Schleifenindex:

```
for i := range []string{"a", "b", "c"}
```

Wird statt des Index nur der Wert benötigt, muss die Index-Variable als zu ignorierende Variable mit einem Unterstrich deklariert werden:

```
for _, v := range []string{"a", "b", "c"} {
```

■ 2.10 Verzweigungen

Für Verzweigungen kennt Go die beiden Schlüsselworte `if` und `switch`. If-Statements bestehen aus einer Bedingung, einem Codeblock sowie einem optionalen `else`-Zweig:

```
1 i := math.Pow(10, 2)
2 if i < j {
3     fmt.Println("i < j")
4 } else {
5     fmt.Println("i >= j")
6 }
```

Die Bedingung wird immer ohne Klammern, der `if`- und `else`-Block immer in geschweiften Klammern angegeben. Der `if`-Bedingung kann ein Statement vorangestellt werden, dessen Ergebnis in der Bedingung verwendet wird:

```
1 if i := math.Pow(10, 2); i < j {
2     fmt.Println("i < j")
3 } else {
4     fmt.Println("i >= j")
5 }
```

Das vorangestellte Statement führt zu einer kompakteren Schreibweise und reduziertem Variablen-Scope. Die Schreibweise wird häufig zum Aufruf von Funktionen mit potenziell fehlerhaftem Ausgang verwendet:

```
1 if b, err := json.Marshal(contact); err != nil {
2     log.Error(err)
3 } else {
4     send(b)
5 }
```

Der Kontakt wird nur gesendet, wenn er fehlerfrei serialisiert werden konnte. Die Variablen *b* und *err* sind nur im if- und else-Branch sichtbar.

Switch-Statements bestehen aus einer Bedingung, gefolgt von einer Liste Case-Blöcken. Ein Case-Block besteht aus einer oder mehreren Anweisungen und endet mit dem Beginn des folgenden Case-Blocks:

```
1 switch r.Method {
2 case "GET":
3     log.Info("GET")
4     handleGetRequest(r)
5 case "POST":
6     ...
7 default:
8     ...
9 }
```

Das Schlüsselwort *break* ist optional und kann verwendet werden, um einen Case-Block vor dessen kompletter Abarbeitung abzuschließen. Das Schlüsselwort *fallthrough* kann am Ende eines Case-Blocks eingesetzt werden, um den nachfolgenden Block in die weitere Auswertung mit einzubeziehen, auch wenn dessen Bedingung nicht erfüllt ist.

Eine weitere Verwendungsart von *switch* ist die Ersetzung der Variablen im Switch-Teil durch einzelne Bedingungen in den Case-Zweigen:

```
1 switch {
2 case height <= 4:
3     fmt.Println("Short")
4 case height <= 5:
5     fmt.Println("Normal")
6 case height > 5:
7     fmt.Println("Tall")
8 }
```

Diese Variante des Switch-Statements wird gerne als besser lesbare Version hintereinandergereihter If-Statements verwendet.

■ 2.11 Methoden

Eine Methode ist eine auf ein Objekt¹ gebundene Funktion. Der Begriff *binden* stammt aus der objektorientierten Programmierung und bedeutet soviel wie dem Objekt eine Nachricht oder einen Auftrag zu senden. Das folgende Beispiel erzeugt einen Kreis und ruft darauf die Methode *draw* auf:

```
1 c := Circle{x:50, y: 50, radius:100}
2 c.draw()
```

Eine Funktion wird zu einer Methode, indem dem Funktionsnamen der Typ vorangestellt wird, auf dem die Methode definiert wird:

```
1 func (c Circle) draw() {
2     graphics.Circle(c.x, c.y, c.radius)
3 }
```

Innerhalb des Methodenrumpfs wird das Objekt, auf dem die Methode arbeitet, durch die dem Typ vorangestellte Variable (hier *c*) repräsentiert.

Die Verwendung von Methoden statt Funktionen ist ein wesentliches Merkmal objektorientierter Systeme. Dabei ist es dort eher die Regel denn die Ausnahme, dass eine Methode einen Seiteneffekt auf dem gebundenen Objekt erzeugt. Dieser Seiteneffekt ist gewünscht, da viele Methoden den Zustand des Objektes ändern sollen. Ein Beispiel für Methoden mit Seiteneffekt ist ein Setter, der einem privaten Attribut einen Wert zuweist:

```
1 func (c Circle ) setX(x int) {
2     c.x = x
3 }
4
5 func main() {
6     c := Circle{}
7     c.setX(50)
8     fmt.Println(c)
9 }
```

```
$ go run main.go
x: 0 y: 0, r: 0
```

Der Setter *setX* weist dem Attribut *x* der *Circle*-Instanz *c* den Wert 50 zu. Der anschließende *Println*-Aufruf soll die Zuweisung verifizieren, gibt aber für *x* statt der erwarteten 50 den Initialwert 0 aus. Der Grund für die nicht erfolgte Zuweisung ist die Call-By-Value-Semantik von Go. Entsprechend arbeiten Funktionen und Methoden auf Kopien ihrer Parameter bzw. Objekte. Gemäß dieser Semantik ist die Methode *setX* seiteneffektfrei, da sie nicht auf dem in *main* instanziierten *c*, sondern auf einer Kopie davon arbeitet. Seiteneffekte werden in Go durch den Einsatz von Pointern erzwungen.

¹ In Go gibt es das Konzept *Objekt* nicht. Die Instanzen eines Typs sind eher Werte. Der Begriff *Objekt* wird synonym für einen Go-Wert benutzt, weil sich *Objekt* besser lesen und sprechen lässt.

■ 2.12 Pointer

Ein Pointer ist eine Variable, die statt eines konkreten Wertes eine Speicheradresse enthält. Diese Adresse zeigt auf die Stelle im Hauptspeicher, an der der eigentliche Wert gespeichert ist. Im folgenden Beispiel enthält *p* die Adresse, an der *x* gespeichert ist:

```
1 func main() {
2     x := 1
3     p := &x
4     fmt.Println(p) // => 0xc420014080
5 }
```

Ein Pointer wird entweder über die Funktion *new* oder wie im Beispiel über den Operator *&* erzeugt. Der Zugriff auf einen von einem Pointer referenzierten Wert erfolgt durch einen vorangestellten Stern, der die Variable dereferenziert:

```
fmt.Println(*p) // => 1
```

Seiteneffekte werden erzwungen, indem Funktionsparameter als Pointer-Variablen, d. h. mit einem dem Typ vorangestellten Stern, deklariert werden:

```
1 func inc(p *int) {
2     *p = *p + 1
3 }
```

Die Funktion *inc* arbeitet jetzt nicht mehr auf einer Kopie der an sie übergebenen Variablen. Stattdessen wird *inc* über den Pointer-Parameter *p* mitgeteilt, an welcher Stelle der zu inkrementierende Wert im Speicher steht. Dieser Wert wird dereferenziert, inkrementiert und an die Speicherstelle zurückgeschrieben. *inc* erzeugt den erzwungenen Seiteneffekt:

```
1 func main() {
2     x := 1
3     p := &x
4     inc(p)
5     fmt.Println(x) // => 2
6 }
```

Methoden werden auf eine Call-By-Reference-Semantik umgestellt, indem dem Typ des gebundenen Objekts ein Stern vorangestellt wird. Auf Pointern deklarierte Methoden werden *Pointer Receiver* genannt:

```
1 func (c *Circle) setX(x int) {
2     c.x = x
3 }
4
5 func main() {
6     c := Circle{}
7     c.setX(50)
8     fmt.Println(c.x) // -> 50
9 }
```

Die Methode *setX* arbeitet nicht mehr auf einer Kopie von *Circle*, sondern auf der in *main* instanziierten Version. Die Variable *c* muss dafür nicht explizit als Pointer deklariert werden. Allein die Deklaration von *setX* als Pointer-Receiver sorgt dafür, dass der Methode ein Pointer auf *c* übergeben wird. Entsprechend können Methoden auf Pointer-Typen die an sie gebundenen Objekte verändern.

Ein Vorteil von der Call-By-Value-Semantik von Go ist, dass Funktions- und Methodenauf-rufe immer seiteneffektfrei sind, sofern sie keinen globalen Zustand verändern. Die Seiteneffektfreiheit führt zu robusteren Programmen, da insbesondere unerwünschte Seiteneffekte vermieden werden. Die Verwendung von Pointern macht Seiteneffekte explizit. Der Code macht deutlich, dass ein Seiteneffekt gewünscht und bewusst programmiert wurde.

■ 2.13 Interfaces

Ein Interface beschreibt die Schnittstelle eines Objekts. Ein Interface sagt, was ein Objekt kann, ohne festzulegen, wie es das tut. Technisch ist ein Interface eine Sammlung von Methoden im Sinne einer Verhaltensbeschreibung:

```
1 type Figure interface {
2     Draw()
3     Erase()
4 }
```

Das Interface *Figure* beschreibt Objekte, die sich selber zeichnen und löschen können. Interfaces können in Variablen-Deklarationen, als Rückgabewerte oder als Funktionsparameter verwendet werden:

```
1 func DrawOnCanvas(f Figure) {
2     f.Draw()
3 }
```

Zur Laufzeit muss sich hinter *f* ein konkretes Objekt verbergen, das sich tatsächlich zeichnen und löschen kann. In Go spricht man davon, dass das Objekt bzw. dessen Typ das Interface *Figure* erfüllen muss. Ein Typ erfüllt ein Interface, wenn er alle Methoden des Interfaces implementiert. Entsprechend erfüllt der Typ *Circle* das Interface *Figure*, wenn er die Methoden *Draw* und *Erase* implementiert:

```
1 type Circle struct {
2 }
3
4 func (c Circle) Draw() { ... }
5
6 func (c Circle) Erase() { ... }
```

Circle erfüllt *Figure*, ohne dass die Implementierung von *Circle* einen Hinweis auf diese Beziehung enthält. *Circle* kann überall dort verwendet werden, wo *Figure*-Instanzen akzeptiert werden:

```
1 func main() {
2     c := Circle{}
3     DrawOnCanvas(c)
4     ...
5 }
```

Die Verwendung von *Figure* entkoppelt *DrawOnCanvas* von *Circle*. Die Funktion *DrawOnCanvas* funktioniert mit allen Typen, die *Draw* und *Erase* implementieren, zum Beispiel *Rectangle* oder *Square*.

Interface-Deklarationen akzeptieren Pointer und Werte, ohne dass ein Interface als Pointer-Variable deklariert werden muss. Entsprechend kann der Funktion

```
DrawOnCanvas(f Figure)
```

sowohl ein Wert als auch ein Pointer übergeben werden. Die Implementierung eines Interface unterscheidet hingegen zwischen Werte- und Pointer-Implementierung. Werden beispielsweise *Draw* und *Erase* als Pointer Receiver implementiert, liefert das folgende Beispiel einen Übersetzungsfehler:

```
1 func (c *Circle) Draw() { ... }
2
3 func (c *Circle) Erase() { ... }
4
5 func main() {
6     c := Circle{}
7     DrawOnCanvas(c)
8     ...
9 }
```

```
$ go build main.go
```

```
./main.go:22:14: cannot use c (type Circle) as type Figure in argument
to DrawOnCanvas: Circle does not implement Figure (Draw method has pointer receiver)
```

Die Fehlermeldung liefert den Grund für das Problem: Der Wertetyp *Circle* implementiert *Figure* nicht mehr. Wird der Funktion *DrawOnCanvas* statt eines Werts ein Pointer auf *Circle* übergeben, lässt sich das Programm wieder übersetzen, da **Circle* das erwartete Interface implementiert:

```
1 func main() {
2     c := &Circle{}
3     DrawOnCanvas(c)
4     ...
5 }
```

■ 2.14 Arrays und Slices

Neben den bekannten Basistypen enthält Go mit Arrays und Slices zwei weitere wichtige Typen.

2.14.1 Arrays

Arrays sind Listen gleichen Typs mit fester Länge.

```
var a [3]string
```

Das Array *a* hat eine Länge von 3 und enthält Elemente vom Typ *string*. Der Zugriff auf Elemente erfolgt indexbasiert:

```
fmt.Println(a[0])
```

Arrays können per Literal erzeugt und initialisiert werden. Das folgende Array-Literal erzeugt ein 4-elementiges String-Array und initialisiert die ersten drei Elemente. Das vierte, nicht initialisierte Element wird mit dem Defaultwert `_` für Strings initialisiert:

```
var a = [4]string{"Kalle", "Ruscha", "Penny"}
```

Die Längenangabe ist optional und kann durch `...` ersetzt werden. Die Länge des Arrays ergibt sich dann aus der Anzahl der initialisierten Elemente:

```
1 var a = [...]string{"Kalle", "Ruscha", "Penny"}
2 fmt.Println(len(a)) // => 3
```

Die Build-In-Funktion *len* liefert die Länge eines Arrays, die zum Beispiel zum Iterieren über die enthaltenen Elemente genutzt werden kann:

```
1 for i := 0; i < len(a); i++ {
2     fmt.Println(a[i])
3 }
```

Eleganter wird die Iteration mit der Build-In-Funktion *range*, die neben dem Wert auch die Index-Position des Elements liefert:

```
1 for i, v := range a {
2     fmt.Println(i, ":", v)
3 }
```

2.14.2 Slices

Die eigentliche Arbeit auf Arrays erfolgt mithilfe von Slices. Eine Slice ist eine dynamische Sicht auf ein Array. Das Array einer Slice wird als das *unterliegende Array* der Slice bezeichnet. Eine Slice wird erzeugt, indem dem unterliegenden Array zwei Indizes *i*, *j* in eckigen

Klammern übergeben werden:

```
1 var a = [...]string{"Kalle", "Ruscha", "Penny", "Lotta"}
2 var s = a[1:3]
3 fmt.Println(s[0]) // => Ruscha
4 fmt.Println(s[1]) // => Penny
```

Die Slice *s* enthält die Elemente Ruscha und Penny des Arrays *a*. Slices haben eine Länge (*len*) und eine Kapazität (*cap*). Die Länge ist die Anzahl der enthaltenen Elemente, die Kapazität die mögliche Obergrenze, bis zu der die Slice erweitert werden kann.

```
1 var s = a[1:3]
2 fmt.Println(len(s)) // => 2
3 fmt.Println(cap(s)) // => 3
```

Die Slice *s* enthält zwei Elemente, hat aber noch Luft für ein weiteres. Übersteigt die Länge einer Slice deren Kapazität, bricht das Programm mit einem *Out of bounds*-Fehler ab.

```
s := a[1:5] // => invalid slice index 5 (out of bounds for 4-element array)
```

Das Slice-Literal *[]type* erzeugt eine Slice sowie ein zugehöriges unterliegendes Array, im Beispiel vom Typ *string* mit einer Länge von 4:

```
var a = []string{"Kalle", "Ruscha", "Penny", "Lotta"}
```

Der einzige Unterschied zwischen Array- und Slice-Erzeugung ist die fehlende Größenangabe in den eckigen Klammern. Alternativ zur Literalschreibweise kann eine Slice mit der Build-In-Funktion *make* erzeugt werden, die eine Parametrisierung von Länge und Kapazität der Slice erlaubt:

```
1 var s = make([]int, 0, 100)
2 fmt.Println(len(s)) // => 0
3 fmt.Println(cap(s)) // => 100
```

Die Erzeugung mit *make* ist dann sinnvoll, wenn von vornherein klar ist, dass die Anzahl der Elemente im unterliegenden Array ansteigend ist:

```
1 var s = make([]int, 0, 100)
2 for i := 0; i < 100; i++ {
3     s = append(s, rand.Int())
4 }
```

Die Build-In-Funktion *append* hängt ein neues Element an die im ersten Parameter übergebene Slice an. Das neue Element wird im unterliegenden Array gespeichert. Dessen Länge entspricht der Slice-Kapazität, im Beispiel also 100. Arrays sind *immutable*² und können nur verändert werden, indem sie kopiert werden. Sobald *append* die Länge des unterliegenden Arrays überschreitet, erzeugt die Funktion ein neues, größeres Array und kopiert die alten sowie das neue Element in das neu erzeugte Array. Dieser Schritt wird vermieden, wenn von vornherein ein ausreichend großes Array erzeugt wird.

² Ein Objekt ist *immutable*, wenn es nach seiner Erzeugung nicht mehr verändert werden kann.

2.14.3 Polymorphe Arrays

Der Typ eines Arrays ist bestimmt durch Länge und Elementtyp. Die Verwendung eines Interface als Array-Typ ermöglicht die polymorphe³ Verwendung des Arrays:

```

1  type Figure interface {
2      Draw()
3  }
4
5  type Circle struct {...}
6
7  func (c Circle) Draw() {...}
8
9  type Rectangle struct {...}
10
11 func (r Rectangle) Draw() {...}
12
13 var figures = [...]Figure{Circle{}, Circle{}, Rectangle{}}
14 for _, f := range figures {
15     f.Draw()
16 }

```

Das Array *figures* weiß zur Laufzeit nicht, ob das jeweilige Element ein Kreis oder Rechteck ist. Das einzig Wichtige ist das Vorhandensein einer *Draw*-Methode.

■ 2.15 Maps

Neben Arrays sind Maps der zweite wichtige Container-Datentyp in Go. Maps enthalten Key-Value-Paare und werden mithilfe des Map-Literals erzeugt:

```

1  var m = map[int]string {
2      1: "Kalle",
3      2: "Penny",
4  }

```

Der in eckigen Klammern angegebene Wert ist der Typ des Keys und der darauffolgende Wert der Typ der Map-Elemente. Alle Elemente einer Map haben denselben Key- und Werte-Typ. Das Beispiel erzeugt eine Map mit Integer-Keys, die Strings enthält. Alternativ zur Literalschreibweise können Maps mit *make* erzeugt werden:

```

1  var m = make(map[int]string, 2)

```

Der optionale Längenparameter bestimmt die Größe der initial allokierten Map. Wird er weggelassen, wird initial eine sehr kleine Map erzeugt.

³ Polymorph bedeutet, dass ein Typ zur Laufzeit verschiedene Formen annehmen kann. Im Beispiel können die *Figure*-Elemente des Arrays die Formen Kreis oder Rechteck annehmen.

Die Zuweisung von Elementen erfolgt durch Angabe des Keys in eckigen Klammern, gefolgt vom Zuweisungsoperator:

```
1 m[1] = "Kalle"
```

Bereits vorhandene Werte werden überschrieben. Der Zugriff auf Map-Elemente erfolgt in umgekehrter Richtung:

```
1 m[1] = "Kalle"
2 fmt.Println(m[1]) // => Kalle
```

Enthält die Map unter dem angegebenen Key keinen Wert, liefert der Zugriff den Defaultwert des jeweiligen Typs:

```
fmt.Println(m[100]) // => ""
```

Das ist nicht immer gewünscht, denn ein leerer String ist ein gültiger Wert und eignet sich nicht für das Prüfen auf das Vorhandensein eines Elements. Aus diesem Grund erfolgt die Überprüfung, ob die Map einen Wert für einen bestimmten Key enthält, über das sogenannte *Ok-Idiom*:

```
1 if v, ok := m[1]; ok {
2     fmt.Println(v) // => Kalle
3 }
```

Das Keyword *range* iteriert durch die Elemente einer Map und liefert für alle Elemente Key und Wert:

```
1 for key, value := range m {
2     fmt.Println(key, value)
3 }
```

■ 2.16 Verzögerungen, Panic und Recover

Eine interessante Variante zur Steuerung des Kontrollflusses ist die Verwendung von *defer*. Defer verzögert einen Funktionsaufruf solange, bis der umschließende Funktionsblock beendet ist:

```
1 func readFile() (int, []byte) {
2     f, err := os.Open("/tmp/contact.json")
3     defer f.Close()
4     check(err)
5     ...
6 }
```

Egal, was in *readFile* nach dem Öffnen der Datei passiert, *f.Close* wird erst am Ende von *readFile* ausgeführt. Die Verwendung von *defer* hat zwei Vorteile: Zum einen werden Aufräumarbeiten direkt an die Stelle gerückt, auf die sie sich beziehen. So lässt sich einfacher erkennen, dass eine geöffnete Datei auch wieder geschlossen wird. Zum anderen werden verzögerte Funktionsaufrufe auch dann ausgeführt, wenn die Funktion in ihrem Verlauf *crashed* (aka *panicked*).

Ein Go-Programm kann aus zwei Gründen panicken: Entweder entdeckt die Go-Runtime einen Laufzeitfehler und beendet das Programm mit Ausgabe eines Stacktrace. Oder das Programm wird explizit durch Aufruf der Build-In-Funktion *panic* beendet. Go-Programmierer benutzen *panic* für Fehler, die quasi unmöglich sind, aber doch irgendwie behandelt werden müssen. Zum Beispiel kann die Serialisierung eines Kontakts eigentlich nicht schiefgehen, aber was, wenn doch?

```

1  if b, err := json.Marshal(contact); err == nil {
2      send(b)
3  } else {
4      panic(err)
5  }
```

Ein *panic*-Aufruf beendet das komplette Programm, was häufig, aber nicht immer gewünscht ist. *Panicked* beispielsweise eine Go-Routine⁴, dann soll das nicht zwingend zum Abbruch der Main-Funktion führen. Mithilfe der Build-In-Funktion *recover* lässt sich auf *panic* reagieren:

```

1  func foo() (err error) {
2      defer func() {
3          if p := recover(); p != nil {
4              err = fmt.Errorf("%v", p)
5          }
6      }()
7      ...
8      panic("help")
9      return nil
10 }
```

Ein *Recover*-Aufruf findet immer in einer verzögert ausgeführten Funktion statt. *Panicked* die umschließende Funktion, dann werden alle verzögerten Funktionen in umgekehrter Reihenfolge ausgeführt. Das verzögerte *Recover* im Beispiel bricht den *Panic*-Aufruf ab und liefert den ursprünglich an *Panic* übergebenen Wert. Dieser wird dem benannten Return-Wert *err* zugewiesen und unmittelbar zurückgeliefert.

Sinnvolle *Recovery*-Fälle sind Aufräumarbeiten vor endgültigem Programmabbruch, Handling von *Panic*-Aufrufen in Third-Party-Bibliotheken oder das Abfangen von *Panic*-Aufrufen aus HTTP-Handlern heraus. Letzteres ist im Go-Package *net/http* so umgesetzt und verhindert ein Beenden des kompletten HTTP-Servers, wenn ein einzelner Request-Handler mit *panic* abbricht.

⁴ Eine Go-Routine ist ein parallel ablaufender Strang innerhalb eines Go-Programms. Go-Routinen werden in Abschnitt 10.4.1 des Kapitels *Skalierung* beschrieben.

■ 2.17 Was sonst noch?

Dieses Kapitel versteht sich als Grundlagenkapitel. Das Kapitel beschreibt gerade soviel Go wie nötig ist, um die folgenden Kapitel zu verstehen. Weitergehende Konzepte wie Closures, Middleware, Concurrency oder Kontexte werden im weiteren Verlauf des Buches immer dann eingeführt, wenn sie konkret benötigt werden.

Stichwortverzeichnis

A

- Accept-Header 32
- Adapter 76, 78, 80
- API 1
- API-Spezifikation 119
- API-Test 93, 94, 103
- append 22
- application/hal+json 117
- Architekturstil 27, 35
- Array 21
 - polymorph 23
- assert 92
- Authentifizierung
 - Challenge 128
 - Client 127
 - HTTP Basic 128
 - HTTP Digest 132
 - JWT 134
 - Server 139
- Authentizität 127
- Autorisierung 127, 141
 - Daten 142
 - Rollen 141

B

- Base64 128, 135
- Bearer-Token 135
- Benutzerdefinierte Typen 13
- bool 13
- Bounded Context 72
- break 16
- Build-In-Funktion
 - append 22
 - cap 22
 - defer 24
 - delete 63
 - len 21, 22
 - make 22, 23
 - panic 25

- range 15
- recover 25

byte 13

C

- Cache
 - Lokal 155
 - Proxy 156
 - Reverse Proxy 156
- Cacheability 36, 156
- Caching 155
 - Bedingte Validierung 158
 - ETags 164
 - Header 158
 - Invalidierung 159
 - TTL 157
 - Validierung 157, 161
- Caching Invalidierung 163
- Call-By-Reference 18
- Call-By-Value 17
- cap 22
- case 16
- Certificate Authority 139
- Channel 151
- Claim 134
- Client-Server-Anwendung 35
- Client-Server-Architektur 27
- Clientstatus 148
- Closure 82, 131
- Code-Smell 84
- complex 13
- const 9
- Constraints
 - Cacheability 36
 - Client-Server 35
 - Code on Demand 37
 - Layered System 36
 - Uniform Interface 36
 - Zustandslosigkeit 35

- Content-Type
 - application/hal+json 113, 117
 - application/json 65, 113
 - application/pdf 65
- Content-Type Header 32
- context 152
- context.WithTimeout 152
- Continuous Integration-Server 105
- Core-Domainmodell 73
- CRUD 43
 - vs. Hypermedia 124

D

- Datenintegrität 127
- DDD 2, 72
- DefaultServerMux 41
- defer 24
- Dekorator 117
 - HTTP-Handler 131
 - Middleware 130
 - Rollen 141
 - schachteln 131, 142
- DELETE 31, 45
- delete 63
- Dependency-Inversion-Prinzip 88
- Deployment 69
- Diffie-Hellman-Verfahren 143
- Direktzuweisung 11
- Domain-driven Design 2, 72
- Domainmodell 53, 73

E

- Einzelne Tests ausführen 106
- ELIZA 75
- else 15
- _embedded 121
- encoding/json 45
- Entkopplung 87
- Entry-URI 112
- .env 129
- ETag 158, 164
- expand 121

F

- Fachdomäne 72
- Fake-Objekt 67, 99
 - FakeRepository 100
 - Unit Test 102
 - vs. Mock-Objekt 102
- fallthrough 16
- Fehlerbehandlung 16
- Filterressource 30

- float 13
- for 14
- func 11
- Funktionen 11
 - init 10
 - main 10
 - TestMain 91

G

- GET 30
- Go
 - Build Tags 105
 - Channel 151
 - Cross-Compilierung 68
 - Eigenschaften 2
 - GOPATH 8
 - Installation 7
 - Middleware 130
 - Modules 8
 - Tool 9
 - Unit Test 90
 - Warum? 2, 68
 - Workspace 8
- go
 - build 41
 - get 42
 - install 9
 - run 9
 - test 91
 - cover 106
 - tags 105
- GOPATH 8
- gorilla/mux 41
- Go-Routine 151
 - Channel 151
 - HTTP-Server 104
- GraphQL 3

H

- HAL 113, 116
 - Dekorator 117
 - _embedded 121
 - HTTP-Methoden 114
 - Links 115
 - _links 113
 - self 114
- HATEOAS 110
 - Client 120
 - Entry-URI 112
 - Response 113
 - Zustandsmodellierung 111
- HEAD 31

Header

- Accept 32, 65, 119
- Authorization 127
- Cache-Control 157, 160, 163
- Content-Type 32
- ETag 158
- Expires 157
- If-Match 66
- If-Modified-Since 66, 158
- If-None-Match 66, 158
- If-Range 66
- If-Unmodified-Since 66
- Last-Modified 158, 160
- Location 33
- Retry-After 33
- WWW-Authenticate 127

Hexagonale Architektur 73

- Bootstrapping 80
- Entkopplung 87
- Layer 73
- Port 74
- Wartbarkeit 87

HTTP Basic 128**HTTP Caching 156****HTTP Digest 132****HTTP-Adapter**

- Cache-Header setzen 160
- Invoice to JSON 80
- JSON to Invoice 78
- Last-Modified-Since 162

HTTP-Authentifizierung 127**http.HandleFunc 40****http.Handler 41****http.ListenAndServe 40****http.ListenAndServeTLS 144****HTTP-Methode 30**

- DELETE 31, 45
- GET 30
- HEAD 31
- idempotent 30
- OPTIONS 31
- PATCH 31
- POST 31, 43
- PUT 31, 44
- sicher 30

http.Request 40**http.ResponseWriter 40****HTTPS 140****http.ServeContent 66****HTTP-Test 94, 102****httptest.NewRecorder 92****httptest.ResponseRecorder 92****Hyperlink 34, 110****Hypermedia 34**

- Client 119
- vs. CRUD 124

Hypermedia Control 27**Hypertext Application Language 113****I****idempotent 30****Identity Provider 135, 137****Idiom**

- Ok 24

if 15**immutable 22, 58****import 10****init 10****Installation 7****int 13****Integrationstest 93, 94, 97****Interface 19, 76, 79****Interface-Segregation-Prinzip 88****ioutil.ReadAll 44****J****JSON 46**

- Arrays 47
- Decoding 48
- Encoding 46
- Maps 47
- Marshal 46
- omitempty 48
- Request-Body decodieren 59
- Unmarshal 59

JSON Web Token 134**json.Marshal 46****json.Unmarshal 48****JWT 134**

- Autorisierung 141
- Bearer-Token 135
- Claim 134
- Erzeugung 135
- Private Key 136
- Public Key 137
- Signatur 136
- Validierung 136

K**Keycloak 136****Konfigurationsdaten 129****Konstruktorfunktion 57, 117****L****Layered System 36**

Layering 72
 len 21, 22
 _links 113
 Liskovsches Substitutions-Prinzip 88
 Listenressource 29
 Literal
 – Array 21
 – Map 23
 – Slice 22
 – struct 13
 Load Balancer 147
 Location-Header 33
 Lose Kopplung 111, 124

M

main 9, 10
 make 22, 23
 – Channel 151
 – Map 23
 – Slice 22
 Man-in-the-Middle-Angriff 134
 Map 23
 Maturity Model 27
 Media Type 32
 Methoden 17
 Middleware 130
 – JWT 137
 Minimal Viable Product 52
 Mockery 101
 Mock-Objekt 67, 101
 – Mock-Repository 101
 – vs. Fake-Objekt 102
 Modules 8
 Multiplexing 41
 mux.Vars 43
 MVP 52
 MySQL 11
 – Repository 79

N

net/http 39
 new 18

O

Objekte 17
 Objektorientierte Programmierung 17
 Open Close-Prinzip 88
 OpenAPI 119
 openssl 139
 OPTIONS 31

P

Package 10
 – context 152
 – crypto/tls 139
 – crypto/x509 139
 – dgrijalva/jwt-go 136
 – encoding/json 45
 – gorilla/mux 41
 – import 10
 – joho/godotenv 129
 – net/http 39
 – net/http/httptest 92
 – stretchr/testify 91
 – testing 90
 – vektra/mockery 101
 panic 25
 PATCH 31
 Pointer 18, 58, 123
 Pointer Receiver 18, 20, 58
 Port 74, 76
 – UpdateInvoicePort 99
 POST 31, 43
 – Buchung zufügen 61
 – Rechnung erstellen 59
 Presenter 85, 118
 – ActivitiesPresenter 160
 – HALInvoicePresenter 122
 – JSONInvoicePresenter 117
 – PDFInvoicePresenter 85
 Primärressource 30
 Programmstruktur 9
 Proxy Cache 156
 PUT 31, 44
 – Rechnung abschließen 64

Q

Quellcode-Abhängigkeit 74, 77

R

range 15, 21, 24
 recover 25
 Regressionen 108
 Repository 57
 Repräsentation 32
 Request-Handler 40
 Resource Expansion 121
 – _embedded 121
 – expand 121
 Ressourcen 29
 – Repräsentation 32
 Ressourcenstatus 149

- REST 27
 - Constraints 35
 - CRUD-Services 43
 - HATEOAS 110
 - Level 0-3 27
 - Maturity Model 27
 - Überblick 27
 - vs. GraphQL 3
 - Warum? 3
- rest.BasicAuth 131
- Restvoice
 - Caching 159
 - CLI 87
 - Kern-Use-Cases 52, 55
 - Kopplung 67
 - Ressourcen 55
 - RESTfulness 68
 - Sicherheit 66
 - Skalierbarkeit 150
 - Testbarkeit 67
 - Unterstützer-Use-Case 52
 - Use Cases 52
 - vertikal skalieren 151
 - Vision 51
- Retry-After-Header 33
- Reverse Proxy Cache 156
- Root-Zertifikat 140
- Routing 40
- Rückgabewerte 11
 - ignorieren 15
 - Namen 11
- S**
- Schichtenarchitektur 72
- Schleife
 - for 14
- Seiteneffekt 17, 58
 - erzwingen 18
- Separation of Concerns 72
- Sichtbarkeit 16
- Sichtbarkeitsregeln 10
- Signatur
 - HS256 134
- Single Responsibility-Prinzip 87
- Sitzungsstatus 148
- Skalierbarkeit 147
 - Clientstatus 148
 - horizontal 147
 - Ressourcenstatus 149
 - Sitzungsstatus 148
 - vertikal 150
 - Zustandslosigkeit 148

- Slices 21
- SOLID 87
- SSL 139
 - Certificate Authority 139
 - Zertifikat 139
- Standardtypen 12
- State Machine 110
- Statische Typisierung 12
- Statuscodes 32
 - 2xx 33
 - 3xx 33
 - 4xx 33
 - 5xx 33
- Strategisches Design 72
- strconv.Atoi 43
- string 13
- struct 13
 - einbetten 14, 46, 117
 - Initialisierung 13
 - schachteln 14
- Subressource 30
 - einbetten 121
- switch 16, 119
 - break 16
 - case 16
 - fallthrough 16

T

- Tag
 - json 47
- Test Coverage 106
- Testabdeckung 94, 108
- Testausführung 105
- Testbarkeit 88
- Testen
 - Coverage 106
 - Externe Systeme 99
 - FakeRepository 100
 - HTTP-Test 92
 - Mock-Objekt 101
 - Rechnung abschließen 94
 - Unit Test 90
 - Was testen? 93
- Testify 91
- testing.Short 106
- testing.T 90
 - testing.T.Error 90
 - testing.T.Fail 90
 - testing.T.FailNow 90
 - testing.T.Fatal 90
 - testing.T.Logf 90
- TestMain 91

- Testpyramide 93
- Testseparation 105
- TestszENARIO 95
- TLS 139, 143
- Typen 12
 - benutzerdefiniert 13
 - standard 12
 - zusammengesetzt 13
- Typinferenz 12

U

- Ubiquitous Language 72
- uint 13
- UI-Test 93
- Uniform Interface 36
- Uniform Ressource Identifier 29
- Unit Test 89, 94
 - AddPosition 97
 - Definition 89
 - Fake-Objekt 102
 - Setup 91
 - Teardown 91
- URI 29
- URL
 - Parameter 59
 - Templates 42
- Use Case 52
 - Buchung löschen 63
 - Buchung zufügen 60
 - Kundenliste anfordern 56
 - Rechnung abschließen 64, 83
 - Rechnung anfordern 85

- Rechnung erstellen 57, 77
- Rechnung zustellen 65
- Tätigkeit buchen 82

V

- var 11
- Variablen 11
 - Export 12
 - ignorieren 15
 - Sichtbarkeit 12
- Vendoring 8
- Verschlüsselung 127, 143
- Verzweigungen
 - if 15
 - switch 16
- Vision 51

W

- Wartbarkeit 87
- Web-Frameworks 2

X

- x509.NewCertPool 140

Y

- YAML 120

Z

- Zusammengesetzte Typen 13
- Zustandsdiagramm 110
- Zustandslosigkeit 35, 148