7 Projekt: Code generieren

7.1 Einleitung

In diesem kleinen Projekt wollen wir ein Tool für das Generieren von Code bauen. Code-Generatoren lassen sich in Go einfach umsetzen und leicht in die Toolchain einbauen. Unser Programm soll einfache Template-Dateien verwenden, um daraus Code zu generieren. Hierfür werden wir das Paket `text/template` aus der Standardbibliothek verwenden.

Anschließend werden wir unseren Code-Generator für ein einfaches Beispiel anwenden. Wir werden zuerst eine einfache Implementierung für die Datenstruktur *Stack* für Integer schreiben. Anschließend werden wir aus unserer Implementierung ein Template für den Code-Generator schreiben.

In einem letzten Schritt bauen wir ein einfaches Programm, das Stacks für unterschiedliche Typen benötigt. Innerhalb des Codes werden wir dann Anweisungen für `go generate` einbauen.

Lernziele

Dieses Projekt verfolgt dabei folgende Lernziele:

- Umsetzung eines *Command Line Interface* inklusive der Möglichkeit, das Benutzer unterschiedliche Parameter setzen können
- Erstellen von Unit-Tests
- Erstellen und Anwendnen eines Text-Templates für das Paket `text/template`
- Einbauen von Anweisungen für `go generate` im Code

7.2 Ein Tool, um Code zu generieren

Als Erstes wollen wir definieren, was unser Programm später exakt tun soll. Der Code-Generator soll es uns ermöglichen, dass wir anhand einer Vorlage Implementierungen für unterschiedliche Typen generieren können. Wenn wir beispielsweise Datenstrukturen für unterschiedliche Typen benötigen, brauchen wir für jeden Typ auch eine eigene Implementierung.

Schauen wir uns dazu einmal die Implementierung der Push-Methode für die Typen `int` und `string` an.

Listing 7–1 *Push-Methode des intStack*

```
type intStack struct {
    stackSlice []int
}

func (s *intStack) Push(in int) {
    s.stackSlice = append(s.stackSlice, in)
}
```

Listing 7–2 *Push-Methode des stringStack*

```
type stringStack struct {
    stackSlice []string
}

func (s *stringStack) Push(in string) {
    s.stackSlice = append(s.stackSlice, in)
}
```

Der Code in den Listings 7–1 und 7–2 unterscheidet sich nur durch den Typ. Dort, wo in Listing 7–1 `int` steht, steht in Listing 7–2 `string`.

Definition unserer Features

Unser Generator soll es uns ermöglichen, die Implementierung der Datenstruktur für jeden beliebigen Typ zu generieren. Dafür soll dieser zusätzlich eine Vorlagedatei verwenden, die für den Typ einen Platzhalter beinhaltet. Unser Programm benötigt bei jeder Ausführung die Vorlagedatei und den Typnamen, für den wir den Code generieren.

Jetzt, da wir wissen, was unser Tool können soll, machen wir uns daran, dem Kind auch einen Namen zu geben. Als Generator für Go-Code wählen wir die Abkürzung `gogen`.

Definition des CLI

Bevor wir uns ans Coden machen, müssen wir noch definieren, mit welchen Argumenten `gogen` aufgerufen wird. Wir wollen unser Beispiel möglichst einfach halten, deshalb soll auch der Aufruf des Tools einfach sein.

```
> gogen [template] [typename]
```

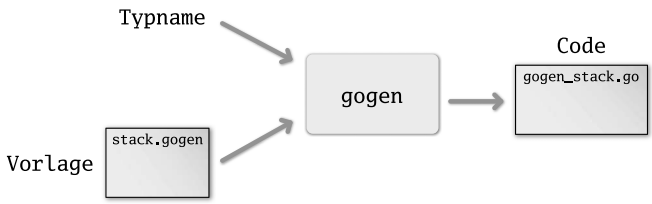

Abb. 7–1
Die Logik von gogen

Als erstes Argument erwarten wir den Pfad zur Vorlage, also dem *Template*. Als zweites Argument soll der Anwender den Typnamen angeben. Für die Vorlagedatei `stack.gogen` wäre der Aufruf wie folgt:

```
> gogen stack.gogen int
> gogen stack.gogen string
```

Jetzt dürfte uns ungefähr klar sein, wo die Reise hingehen soll. Wir legen für unser Projekt ein Verzeichnis mit dem Namen `gogen` an. Anschließend gehen wir über einen Terminal in unser neues Verzeichnis und initialisieren das Projekt.

```
> go mod init example.com/gogen
```

Als URL haben wir `example.com/gogen` gewählt, da wir eine URL angeben müssen. Da unser Tool als Übung vorerst nur lokal auf unserem Rechner existieren soll, benötigen wir dafür auch keine Versionsverwaltung über beispielsweise GitHub.

In unserem Projekt legen wir nun die Datei `main.go` an. In einem ersten Schritt beginnen wir mit dem Ausprägen des Nutzer-Interfaces.

Listing 7–3
Prüfen der Eingabeparameter

```
package main

import (
    "fmt"
    "os"
)

func main() {
    if len(os.Args) != 3 {
        fmt.Println("gogen benötigt zwei Argumente")
        fmt.Println("gogen [template] [typename]")
        os.Exit(1)
    }
}
```

os.Args für das Auslesen der Eingabeparameter

Um die Eingabeparameter des Programms zu verarbeiten, verwenden wir das Paket `os`. Dieses Paket beinhaltet das Slice `os.Args`, das

für den Index 0 `os.Args[0]` den Programmpfad und ab Index 1, also `os.Args[1:]`, die Argumente des Aufrufs enthält. Für unsere Schnittstelle erwarten wir immer zwei Argumente. Mit dem Programmpfad muss dieses Slice somit drei Einträge enthalten. Im Code prüfen wir einfach die Größe/Länge von `os.Args`. Sollte diese nicht drei Einträgen entsprechen, brechen wir mit `os.Exit()` ab. Dieser Funktion müssen wir noch einen Rückgabewert für den *Exit Status* mitgeben. Dabei steht 0 für *erfolgreiche Ausführung* und alle anderen Werte für einen Fehler. Deshalb übergeben wir 1, da wir ja einen fehlerhaften Aufruf abgefangen haben. Zusätzlich geben wir dem Anwender noch eine kleine Hilfe an die Hand, wie `gogen` richtig zu verwenden ist.

Jetzt testen wir unser Programm, indem wir einfach drei Argumente übergeben. Als Ergebnis erwarten wir eine Fehlermeldung. Für diesen Test verwenden wir `go run`, da wir damit das Programm nicht kompilieren müssen.

Listing 7–4 *Aufruf mit drei Argumenten*

```
> go run main.go abc def ghi
gogen benötigt zwei Argumente
gogen [template] [typename]
exit status 1
```

Die gleiche Ausgabe erhalten wir auch, wenn wir nur ein Argument oder noch mehr Argumente beim Aufruf mitgeben. Jetzt ist sichergestellt, dass wir immer zwei Argumente beim Aufruf bekommen und später auf `os.Args[1]` und `os.Args[2]` zugreifen können.

Listing 7–5 *Argumente auslesen*

```
templateFileName := os.Args[1]
typeName := os.Args[2]
```

Wir weisen nun den beiden Argumenten sprechende Variablen zu, damit wir später im Code verstehen, um welchen Wert es sich handelt. Technisch sind diese Variablen natürlich nicht notwendig, jedoch ist die Variable `templateFileName` später im Code um einiges besser zu verstehen als `os.Args[1]`.

sprechende Variablen

Im nächsten Schritt kümmern wir uns um das Verarbeiten der Template-Datei. Dafür verwenden wir das Paket `text/template`. An der Stelle ist es wichtig zu wissen, dass die Standardbibliothek zwei Template-Pakete enthält. Das Paket `html/template` ist für das Generieren von HTLM-Dateien zuständig. Das Paket funktioniert eigentlich ähnlich wie `text/template`, nur dass Strings hier besonders behandelt werden. Damit eine *Code-Injection* durch einen Anwender einer Seite nicht möglich ist, werden entsprechende Sonderzeichen durch ein *Auto-Escape* ersetzt. Für unser Programm müssen wir also prüfen, ob hinter den Imports das Paket `text/template` steht.

Bevor wir uns mit der Verwendung von Templates beschäftigen, sollten wir uns noch kurz überlegen, was bei deren Verwendung innerhalb des Pakets passiert. Intern wird das Template geparst. Dabei wird die Datei Schritt für Schritt durchgegangen und als Baum intern abgebildet. Diese Abbildung als *Syntax Tree* ermöglicht dann später beim Ausführen ein schnelleres Zusammenbauen der einzelnen Bausteine inklusive der Platzhalter für die Variablen. Wir benötigen bei der Verwendung also einmal das Parsen des Templates und danach das Ausführen mit den Werten aus den Variablen.

Funktionsweise von Templates

Schauen wir uns aber die einzelnen Schritte direkt im Code an. Das Parsen des Templates erfolgt über die Funktion `template.ParseFiles()`.

Listing 7–6
Template-Datei parsen

```
t, err := template.ParseFiles(templateFileName)
if err != nil {
    fmt.Printf("Fehler beim Parsen: %v\n", err)
    os.Exit(1)
}
```

Damit ein Template erfolgreich geparst werden kann, muss es syntaktisch korrekt sein. Ist es das nicht, kommt es zu einem Fehler. Diesen Fehler fangen wir in unserem Code ab und geben ihn aus. Wenn das Template nicht korrekt ist, brechen wir unser Programm mit `os.Exit(1)` ab. Denn in diesem Fall können wir natürlich auch keinen Code mehr generieren.

Listing 7–7
Output-Datei erzeugen

```
outName := fmt.Sprintf("gogen_%s_gen.go", typeName)
fd, err := os.OpenFile(outName, os.O_CREATE|os.O_WRONLY, 0666)
if err != nil {
    fmt.Println("Fehler beim Erzeugen des Zielfiles: ", err)
    os.Exit(1)
}
defer fd.Close()
```

Jetzt, da wir das Template intern schon mal als Baum in unserem Arbeitsspeicher abgelegt haben, kümmern wir uns um die Ausgabedatei für den generierten Code. Hierfür erzeugen wir mit `os.OpenFile` eine Datei. Die Flags `os.O_CREATE|os.O_WRONLY` sind notwendig, damit die Datei angelegt wird und wir auch in diese Datei schreiben können. Mit `0666` definieren wir die Rechte für die angelegte Datei.

Um den Code einfach zu halten, wählen wir einen Dateinamen, der nur den Typnamen als Variable enthält. Für das Zusammenbauen des Namens verwenden wir `fmt.Sprintf()`, das einen formatierten String erzeugen kann. Anhand des Formatstrings `gogen_%s_gen.go` können wir schnell erkennen, nach welcher Logik die Dateinamen zusammenge-

baut werden. Damit unsere Datei am Ende geschlossen wird, verwenden wir `defer` zusammen mit `fd.Close()`.

Listing 7–8
Daten in eine Struktur schreiben

```
data := struct {
    T string
}{
    typeName,
}
t.Execute(fd, data)
```

Als letzten Schritt erzeugen wir eine Struktur, die den Typnamen beinhaltet. Innerhalb des Templates sind alle Parameter dieser Struktur verwendbar. Da `gogen` nur den Typnamen unterstützt, schreiben wir hier auch nur einen Parameter, den wir mit `T` so kurz wie möglich halten. Da die Struktur nur einmal an dieser Stelle gebraucht wird, verwenden wir dafür eine anonyme Struktur (siehe Kapitel 2.7). Danach führen wir unser Template mit den Daten aus. Die Methode `Execute()` benötigt als Ziel einen `io.Writer`. Hierfür übergeben wir dann `fd`, der als Filedeskriptor auch das `io.Writer`-Interface implementiert.

Alles in allem ist in Listing 7–9 die komplette `main()`-Funktion dargestellt.

Listing 7–9
Der komplette Code von gogen

```
func main() {
    if len(os.Args) != 3 {
        fmt.Println("gogen benötigt zwei Argumente")
        fmt.Println("gogen [template] [typename]")
        os.Exit(1)
    }
    templateFileName := os.Args[1]
    typeName := os.Args[2]

    t, err := template.ParseFiles(templateFileName)
    if err != nil {
        fmt.Printf("Fehler beim Parsen: %v\n", err)
        os.Exit(1)
    }

    outName := fmt.Sprintf("gogen_%s_gen.go", typeName)
    fd, err := os.OpenFile(outName, os.O_CREATE|os.O_WRONLY, 0666)
    if err != nil {
        fmt.Println("Fehler beim Erzeugen des Zielfiles: ", err)
        os.Exit(1)
    }
    defer fd.Close()
```

```
    data := struct {
        T string
    }{
        typeName,
    }
    t.Execute(fd, data)
}
```

Mit `go install` können wir nun unser Tool installieren, sodass es systemweit verwendet werden kann. Der Code ist dabei schlank und kurz, da die meiste Funktionalität durch die Standardbibliothek übernommen wird.

7.3 Template erstellen

Unser Generator ist jetzt fertig, jedoch konnten wir bislang noch nicht testen, ob alles so funktioniert, wie wir uns das gedacht haben. Dafür benötigen wir jetzt noch ein Template. Hierfür wählen wir die Datenstruktur Stack aus. Ein Stack benötigt die Methoden `Push` und `Pop`.

Stack

Ein Stack ist eine einfache Datenstruktur, dir es ermöglicht, Daten geordnet abzulegen. Die Daten werden dabei in einem *Stapel* abgelegt. Mit dem Befehl `Push` können Daten auf den Stapel abgelegt werden. Mit dem Befehl `Pop` wird das oberste Element des Stapels zurückgeliefert und dabei auch vom Stapel entfernt.

Bevor wir die Vorlage für den Generator erstellen, wollen wir einen Stack für den Typ `int` implementieren. Später verwenden wir diesen Code für das Erstellen des allgemeinen Templates. Dafür erzeugen wir ein Verzeichnis `stack` innerhalb unseres Projekts und legen dort eine Datei `int.go` an.

Listing 7–10
Implementierung von intStack

```
package stack

type intStack struct {
    stackSlice []int
}

func (s *intStack) Push(in int) {
    s.stackSlice = append(s.stackSlice, in)
}
```

9 Concurrency Patterns

Das Konzept der nebenläufigen Programmierung kann am Anfang ungewohnt sein, da die Anwendung ein Umdenken zur sequentiellen Programmierung erfordert. Dies bedarf vielleicht zuerst ein wenig Übung. Wenn wir dann einmal die ersten nebenläufigen Programme erstellt haben, werden wir feststellen, dass sich alles sehr schnell wiederholt. Denn eigentlich geht es immer darum, Daten über Channels zu schicken und diese dann innerhalb von Goroutinen zu verarbeiten oder mit Goroutinen andere Goroutinen zu steuern. Wenn wir Nachrichten aus Channels bekommen, steht immer ein Ereignis dahinter, auf das wir reagieren müssen. Dafür gibt es eine Handvoll Patterns, die uns das Leben am Anfang sehr erleichtern. Die wichtigsten Patterns sollen in diesem Kapitel vorgestellt werden.

9.1 Checkliste zu Goroutinen

- Was müssen wir beachten, wenn wir eine Goroutine starten?

Bevor wir mit konkreten Patterns tiefer in die Konstruktion von Goroutinen einsteigen, schauen wir uns ein paar Grundsätze an. Diese Grundsätze sollen uns am Anfang die Arbeit erleichtern.

Bei der Konstruktion von Goroutinen sollten wir uns immer folgende Fragen stellen:

- Wie lange soll die Goroutine laufen?
- Wie wird sie beendet?
- Welche Elemente blockieren den Programmfluss?
- Kann es zu einer Leaking Goroutine kommen?
- Was passiert auf der anderen Seite der Channels?

Gehen wir die Punkte noch einmal kurz durch. Die Frage nach der Laufzeit und dem Beenden der Goroutine muss immer beachtet werden. Hier ist das Context-Pattern (Kapitel 9.4) unser wichtigstes Werkzeug. Denn bei Services, die lange und stabil laufen müssen, ist es wichtig, einmal gestartete Goroutinen gezielt beenden zu können. Wir müssen da-

bei aber aufpassen, dass die Goroutinen auch wirklich beendet werden. Deshalb sollten wir beim Start der Goroutine auch prüfen, dass keine Elemente ungewollt blockieren. Denn blockierende Elemente können zu Leaking Goroutines führen. Allgemein müssen wir bei der Verwendung von Channels auch immer die andere Seite im Hinterkopf behalten.

Leaking Goroutines

Leaking Goroutines sind Goroutinen, die während der Laufzeit nie mehr beendet werden. Bei Programmen, die nur kurz laufen, ist dies nicht so schlimm, da mit dem Beenden des Hauptprogramms auch alle Goroutinen beendet werden. Bei Servern sieht dies jedoch ganz anders aus. Hier kann die Laufzeit mehrere Monate betragen, bis der Server neu gestartet wird. Wenn pro Request eine Goroutine als Leiche im Arbeitsspeicher vorgehalten werden muss, geht früher oder später der Arbeitsspeicher aus.

Für die Prüfung unserer Channels gibt es ein paar Merksätze, die die Funktion von Channels zusammenfassen:

- Daten über einen nil-Channel zu senden, führt zu einem Deadlock.
- Daten aus einen nil-Channel lesen zu wollen, führt zu einem Deadlock.
- Einen nil-Channel zu schließen, führt zu einer Panik.
- Daten über einen geschlossenen Channel zu senden, führt zu einer Panik.
- Daten aus einem geschlossenen Channel zu lesen, liefert den Nullwert des Typs zurück.
- Einen bereits geschlossenen Channel zu schließen, führt zu einer Panik.

Mit diesen Merksätzen können wir nun loslegen, um die einzelnen nebenläufigen Patterns kennenzulernen.

9.2 Goroutinen melden, wenn sie fertig sind

- Wie kann eine Goroutine mitteilen, dass sie mit ihrer Arbeit fertig ist?
- Warum ist das wichtig?

Wenn wir eine Goroutine starten, wird diese aus dem aktuellen Programmfluss herausgenommen und blockiert nicht das Hauptprogramm. Damit wir jedoch mitbekommen, wann eine Goroutine fertig ist, muss diese kommunizieren, wann sie fertig ist. In dem Fall können wir Goroutine und Hauptprogramm wieder synchronisieren.

Wenn eine Goroutine ein Ergebnis ermittelt, wird dieses am Ende der Goroutine über einen Channel gesendet. Da Channels blockieren, werden die Goroutinen auf Sender- und Empfängerseite synchronisiert. Entweder der Empfänger oder der Sender wartet, bis das Ergebnis übermittelt werden kann. Um einer Leaking Goroutine vorzubeugen, sollte der `result`-Channel einen Puffer der Größe 1 besitzen. Denn sollte aus irgendwelchen Gründen kein Empfänger für das Ergebnis mehr zur Verfügung stehen, würde die Goroutine nie beendet werden.

Listing 9–1 *Ergebnis an einen Channel*

```
result := make(chan resultType, 1)
// ...
go func() {
    r := doSomething()
    result <- r
}()
// ...
// Verwende das Ergebnis
ergebnis := <-result
```

Goroutinen können aber auch aktiv melden, dass sie fertig sind. Die einfachste Lösung hierfür ist die Verwendung der `sync.WaitGroup`.

Listing 9–2 *Verwendung der sync.WaitGroup*

```
// WaitGroup
var wg &sync.WaitGroup
go func() {
    doSomething()
    wg.Done()
}()
wg.Wait()
```

Die Verwendung der WaitGroup ist komfortabel, schnell und sicher. Als Variablenname hat sich `wg` durchgesetzt. Wenn möglich, sollten wir diesen Namen also ausschließlich für WaitGroups verwenden.

Wenn wir keine WaitGroup verwenden möchten, bietet sich auch ein Channel an. Diesen schließen wir, sobald die Goroutine fertig ist.

Listing 9–3
Synchronisierung mit einem done-Channel

```
// Channel schließen
done := make(chan struct{})
// ...
go func() {
    doSomething()
    close(done)
}()
// Programmfluss blockiert hier, bis
// done geschlossen wird.
<-done
```

Anstatt den Channel zu schließen, können wir auch eine Nachricht schicken. Dabei sollten wir wieder einen Puffer setzen, denn so sind wir unabhängig von einem Empfänger. Die Goroutine wird in jedem Fall beendet. Der einzige Wermutstropfen ist, dass keine Synchronisierung stattfindet.

Listing 9–4
Erledigt über einen buffered Channel

```
done := make(struct{}, 1)
go func() {
    doSomething()
    done <- struct{}{}
}()
```

Alle der hier vorgestellten Pattern ermöglichen es einer Goroutine, zu kommunizieren, dass sie fertig ist.

9.3 Beenden von Goroutinen

- Wie lassen sich Goroutinen so aussteuern, dass wir sie kontrolliert beenden können?

Noch einmal der wichtigste Grundsatz für Goroutinen als Wiederholung von Kapitel 9.1: Beim Starten einer Goroutine muss auch immer deren Ende berücksichtigt werden. Bezüglich der Laufzeit gibt es zwei unterschiedliche Typen von Goroutinen:

- Goroutinen, die einen Loop beinhalten und auf ein oder mehrere unterschiedliche Ereignisse reagieren und somit *unendlich* laufen, und
- Goroutinen, die eine oder mehrere Anweisungen sequentiell abarbeiten und dann beendet werden.