Kapitel 1
Einführung

In diesem Kapitel werden Sie Microservices und die Programmiersprache Go kennenlernen. Neben den Vorzügen von Go werden Sie erfahren, warum es vorteilhaft ist, Go für Ihre Microservice-Implementierungen zu verwenden.

1.1 Was sind Microservices?

Über lange Zeit wurden Enterprise-Anwendungen in Form von großen zusammenhängenden Einheiten entworfen und umgesetzt. Dieser Ansatz wird als monolithischer Architekturstil bezeichnet. Hier werden sämtliche Geschäftsfunktionalitäten in einer Einheit ausgeliefert und zur Laufzeit innerhalb eines (Betriebssystem-) Prozesses ausgeführt.

Die interne Strukturierung einer solchen Anwendung findet über Mittel der eingesetzten Programmiersprache in Form von Aufteilung in Klassen, Funktionen oder Namensräume statt.

Mit diesem Ansatz wurden und werden viele erfolgreiche Produkte bzw. Projekte umgesetzt. Allerdings nehmen mit zunehmender Größe und Projektlaufzeit der Anwendung auch oftmals die Probleme und damit Frustration über die Anwendung bei Anwendern, Fachabteilungen und Entwicklern zu.

1.1.1 Probleme bei monolithischen Architekturen

Zu den größten Problemen zählen die folgenden:

Neue Features einzubauen ist komplex und fehleranfällig

Die Umsetzung eines Features erfordert immer eine Änderung in der Komplettanwendung. Dies bedeutet, dass vor einer Auslieferung die gesamte Anwendung komplett getestet werden muss. Nebeneffekte können nie ausgeschlossen werden, und eine kleine Änderung z. B. an den Timeout-Einstellungen für den Transaktionsmanager, die in einem Anwendungsteil benötigt wird, kann unter Umständen an einer ganz anderen Stelle in der Anwendung zu massiven Problemen führen. Oftmals sind Abhängigkeiten in der Anwendung nicht völlig transparent und machen Änderungen fehleranfällig. Automatisierte Tests können zwar die Problematik entschärfen,

aber leider nicht komplett aus der Welt schaffen. In vielen Systemen wird aus diesen Gründen angebaut und nicht angepasst, und dies führt wiederum zu komplexeren Systemen, die den Einbau von Features noch weiter erschweren.

Weiterentwicklung der Architektur ist schwierig

Monolithische Anwendungen sind oftmals über Jahre gewachsen, und die Architektur der Anwendung wurde zu Beginn festgelegt. Aus der schlanken Neuentwicklung wurde mit der Zeit eine schwerfällige Legacy-Anwendung. Die zu Beginn getroffenen Architektur- oder Technologieentscheidungen haben sich meist tief in das System eingegraben, und entsprechende Überarbeitungen im Falle einer Revidierung dieser Entscheidungen müssen oftmals an vielen Stellen, auch gleichzeitig, vorgenommen werden. Kluge Architekturentscheidungen wie die strikte Trennung zwischen Technologie und Fachlichkeit oder sinnvolle Abstraktionsschichten erleichtern in den meisten Fällen das Vorgehen, grundlegende Anpassungen bedeuten jedoch immer erheblichen Aufwand und ein gewisses Fehlerrisiko.

Skalierung zur Laufzeit

Eine Skalierung ist bei Monolithen nur für die komplette Anwendung möglich und macht diese unter Umständen sehr komplex, da nicht ausschließlich diejenigen Teilbereiche skaliert werden können, die tatsächlich mehr Ressourcen benötigen. Der technische Aufwand, um z. B. Caches und Synchronisationsmechanismen korrekt umzusetzen und zu testen, darf nicht unterschätzt werden. Skalierung bei Monolithen bedeutet immer, die Anwendung ein weiteres Mal auszuführen und die Last über einen vorgeschalteten Load-Balancer zu verteilen.

Skalierung bei der Entwicklung

Arbeiten mehrere Teams an derselben Codebasis eines Monolithen, muss oftmals viel Abstimmungsaufwand betrieben werden, um die Teams untereinander zu koordinieren. Welches Feature soll oder muss z. B. zu welchem Zeitpunkt ausgeliefert werden? Welche Änderung wirkt sich auf welche anderen Bereiche aus?

Lange Deployments

Monolithische Anwendungen besitzen beim Deployment immer die Komplexität des Gesamtsystems, da auch genau dieses deployt wird. Bei jedem Deployment müssen sämtliche interne Bestandteile sowie externe Abhängigkeiten konfiguriert und gestartet werden. Wird für einen Monolithen z. B. ein IBM WebSphere Application Server eingesetzt, kann der Start der Umgebung gut und gerne 30–60 Minuten dauern. Pro Instanz. Verwaltet die Anwendung zusätzlich einen Status pro Client, der nicht über die Instanzen repliziert wird, muss beim Herunterfahren eventuell gewartet werden, bis sich alle Clients abgemeldet haben bzw. ihre Session ausgelaufen ist.

Lange Build- und Release-Zyklen

Das Erstellen einer monolithischen Anwendung kann sehr viel Zeit in Anspruch nehmen, da sämtliche Codegenerierungsläufe ausgeführt werden müssen und sämtlicher Quellcode zu übersetzen ist. Anschließend müssen die kompletten Unit-Tests der Anwendung durchlaufen werden. Je größer die Codebasis ist, desto länger dauert die Erstellung eines Anwendungsartefakts, das außerdem vor einem Release noch den kompletten Integrations- und Regressionstests mit fachlichen Abnahmen unterzogen werden muss. Alles in allem ein langwieriger und fehleranfälliger Prozess, der oftmals dazu führt, dass weniger Releases durchgeführt und die Release-Zyklen immer länger werden. Eine agile Vorgehensweise lässt sich in einem solchen Umfeld nur schwer umsetzen.

Geringe Innovationskraft

Durch den recht engen und unflexiblen Rahmen bei der Entwicklung eines Monolithen ist es schwer, neue Ideen auszuprobieren und umzusetzen. Wie bereits beschrieben, sind Anpassungen an der Architektur, die für manche neuen Ideen nötig sind, nur sehr schwer umsetzbar. Eine Migration von z. B. einer rein HTML-basierten Anwendung, die ihre Seiten serverseitig generiert, zu einem AJAX/JavaScript-basierten Frontend wird erst dann möglich, wenn die technischen Voraussetzungen zur Umsetzung von REST-Services gegeben sind. Dass keine REST-Service-Unterstützung vorhanden ist, erscheint im ersten Moment vielleicht völlig abwegig, aber bei über Jahre gewachsenen Monolithen ist das Vorhandensein dieser Services nicht selbstverständlich. Eventuell verhindern auch alte Bestandteile der Anwendung ein Update auf eine neue Laufzeitumgebung, welches allerdings für bestimmte Features zwingend notwendig ist.

Herausforderndes Cloud Deployment

Zahlreiche Unternehmen stellen aus den verschiedensten Gründen von selbst betriebenen Rechenzentren mit on-Premise-Lösungen auf in der Cloud gehostete Systeme um. Um das volle Potential einer Cloud-Umgebung nutzen zu können, sind Anpassungen an der Anwendung nötig, die, wie bereits beschrieben, zum Teil nur schwer umsetzbar sind. Es ist auch denkbar, dass hohe Lizenzgebühren einer Laufzeitumgebung Deployment oder Ausführung einen Monolithen in der Cloud verhindern.

Auch die Dateigröße der Anwendung kann bei der Arbeit in einer Cloud-Umgebung von Nachteil sein.

Sicherstellung der internen Anwendungsarchitektur

Je größer die Codebasis ist und je mehr Entwickler an ihr arbeiten, desto schwieriger wird es, die interne Anwendungsarchitektur im Griff zu halten. Auch passen nicht alle

Architektur- oder Programmiervorgaben auf alle Problemstellungen, die während der Entwicklung eines Projektes auftreten. Bei Monolithen wird aus diesem Grund oft versucht, »Wildwuchs« über komplexe Architekturüberwachungswerkzeuge oder durch erzwungene Projektstrukturen zu unterdrücken.

Die Einarbeitung neuer Entwickler in die komplette, bestehende und über mehrere Code-Generationen gewachsene Anwendung ist meist aufwändig.

1.1.2 Gemeinsame Eigenschaften von Microservices

Um alle oben beschriebenen Herausforderungen anzugehen, haben viele Projektteams damit begonnen, ihre monolithischen Systeme aufzulösen und durch kleinere Anwendungen zu ersetzen bzw. neue Bestandteile als separate Applikationen zu entwickeln.

James Lewis und Martin Fowler haben im Jahr 2012 versucht, die Gemeinsamkeiten der verschiedenen Ansätze zusammenzutragen und das entstandene Modularisierungskonzept zu beschreiben. In der Folge ist der Begriff *Microservices* entstanden. Eine einheitliche Definition existiert allerdings nicht. Wenn Sie den Architekturansatz verfolgen möchten, können Sie sich an den nachfolgend beschriebenen Eigenschaften orientieren.

Adrian Cockroft, der bei Netflix maßgeblich bei der Umstellung auf eine Microservice-Architektur beteiligt war, beschreibt den Architekturansatz beispielsweise so:

> *Service-oriented architecture composed of loosely coupled elements that have bounded contexts – Adrian Cockcroft*

Komponenten als separate Services

In der Softwareentwicklung hat Modularisierung und somit die Gliederung eines Systems in kleine, beherrschbare Einheiten schon immer eine sehr große Rolle gespielt. Jede Programmiersprache bietet entsprechende Mechanismen, um einzelne Komponenten oder Module definieren bzw. erstellen zu können.

Für den Begriff *Komponente* steht allerdings keine einheitliche Definition zur Verfügung, und ich möchte mich hier an die 1996 auf der European Conference on Object-Oriented Programming (ECOOP) erarbeitete Beschreibung anlehnen.

Komponenten:

- sind eine zusammengestellte Software-Einheit
- besitzen eine klar definierte Schnittstelle
- können unabhängig voneinander ausgeliefert werden
- sind Baustein für Dritte

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*
> *– ECOOP 1996*

Im Gegensatz zu monolithischen Anwendungen, die in der Regel ebenfalls aus mehreren Komponenten zusammengesetzt sind bzw. sein sollten, wird bei Microservices jede Komponente als separater Service und somit als separater Prozess ausgeführt. Die Kommunikation zwischen den einzelnen Services erfolgt über leichtgewichtige Protokolle wie z. B. REST über HTTP. Die komplette Anwendung besteht somit aus einem Netz aus miteinander kommunizierenden Services.

Diese Aufteilung der Anwendung und die daraus nötige Kommunikation über eine klar definierte Schnittstelle fördern zusätzlich die Kapselung der einzelnen Komponenten. Eine Umgehung der Komponentengrenzen bzw. ihrer definierten Schnittstelle ist in Monolithen eventuell noch möglich, bei Microservices nicht mehr. Die Technik schränkt hierbei im positiven Sinne ein.

Da die Kommunikation zwischen den Services über Remote-Protokolle stattfindet, werden einzelne Aufrufe auch zeitintensiver und müssen eventuell zusammengefasst werden. Schnittstellen von Microservices sollten Sie nicht zu feingranular aufbauen, um keine Laufzeitprobleme zu riskieren.

Steht jede Komponente als separater Service zur Verfügung, können diese Services auch unabhängig voneinander ausgetauscht und aktualisiert werden. In Monolithen können Sie zwar auch einzelne Komponenten austauschen, aber die Aktualisierung in Ihrer laufenden Anwendung erfordert anschließend immer ein Deployment der kompletten Anwendung.

Um eine Business Funktionalität herum gebaut – Bounded Context

In vielen Unternehmen haben sich Unternehmensstrukturen parallel zu den eingesetzten Technologien entwickelt. Das Datenbank-Team kümmert sich um die Datenbanken, Applikationsserver-Teams behandeln die Logik in den Applikationsservern, und das Frontend-Team übernimmt die Arbeit an einer ansprechenden Oberfläche der Anwendungen. Änderungen, die mehrere dieser Technologiebereiche betreffen, erfordern immer Koordinationsaufwand. Wenn Sie z. B. in einem Web-Formular ein neues Feld aufnehmen möchten, das am Ende in der Datenbank gespeichert werden soll, müssen alle drei oben genannten Teams aktiv werden.

In der Folge wird oftmals versucht, Anforderungen direkt im eigenen Zuständigkeitsbereich zu lösen. Das Frontend-Team könnte sich die entsprechenden Daten aus dem Beispiel in einer eigenen Datenhaltung speichern, oder aber das Datenbank-Team setzt direkt Logik in der Datenbank um. Das Ergebnis ist ein System, dessen Anwendungslogik auf viele einzelne Stellen verteilt ist.

Melvin Conway hat 1968 festgestellt, dass die Struktur von Systemen durch die Kommunikationsstruktur von Unternehmen, die diese Systeme erstellen, vorgegeben ist. Seine Feststellung wurde als *Conways Law* bekannt, und Sie können diese Aussage in obigem Beispiel gut erkennen.

> *Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure. – Melvin Edward Conway*
> (Quelle: http://www.melconway.com/Home/Committees_Paper.html)

Microservices sind hingegen crossfunktional und setzen eine Businessfunktionalität durchgängig um, im Idealfall vom Frontend bis zur Persistenz-Schicht. Im Umkehrschluss bedeutet dies auch, dass jede Businessfunktionalität innerhalb eines eigenen Service implementiert wird. Die Grundidee wird oftmals mit der *UNIX-Philosophie* verglichen:

> *Many UNIX programs do quite trivial things in isolation, but, combined with other programs, become general and useful tools. [...]* ***Do one thing and do it well!*** *– UNIX-Philosophie*

Für die Organisation von fachlichen Funktionalitäten innerhalb einer Anwendung und eine Zuordnung zu einzelnen Services ist das *Bounded-Context*-Konzept aus dem *Domain Driven Design* (DDD) hilfreich, das von Eric Evans 2003 durch sein gleichnamiges Buch maßgeblich geprägt wurde.

Ein *Bounded Context* beschreibt einen begrenzten Bereich, in dem ein fachliches Domänenmodell mit zugehöriger Geschäftslogik eingesetzt wird. Stellen Sie sich z. B. einen Web-Shop vor, in dem verschiedene Artikel bestellt werden können:

Eine Komponente ist für die Suche und Anzeige der Artikel zuständig und benötigt Bilder, Rezensionen und Beschreibungstexte für die anzuzeigenden Artikel. Bei der Bestellabwicklung dann sind für diese Artikel zum Teil abweichende Informationen wichtig. Die Komponenten »Suche/Anzeige« und »Bestellabwicklung« haben wenig fachliche Überschneidungen und können über zwei getrennte Domänenmodelle beschrieben werden. Somit besitzt jede dieser Komponenten einen eigenen *Bounded Context*.

In einer Microservice-Architektur bietet es sich an, die Services anhand der fachlich bestimmten *Bounded Contexts* zu schneiden. Dies hat den Vorteil, dass fachliche Anforderungen im Idealfall jeweils in nur einem Service umgesetzt werden müssen und auch unabhängig voneinander ausgeliefert werden können.

Die Definition und Abgrenzung eines *Bounded Contexts* ist nicht immer einfach, und der Begriff **Micro**service verleitet Teams oftmals dazu, sehr viele, sehr kleine Services zu implementieren.

Lassen Sie sich hier von einer weiteren Aussage von Martin Fowler leiten:

> *Fowler's First Law of Distributed Objects: Don't distribute your objects.*

Lassen Sie zusammengehörige Funktionalität zusammen und definieren Sie immer zuerst ihren *Bounded Context*.

Smart Endpoints and Dump Pipes

In der Microservice-Welt hat sich der Ansatz von »smart endpoints and dump pipes« verbreitet. Die Idee dahinter besteht darin, die einzelnen Endpunkte, also die Services, mit allem auszustatten, was sie benötigen, und sie gleichzeitig so weit wie möglich von anderen Services zu entkoppeln. Die Kommunikation zwischen den Services soll möglichst leichtgewichtig sein und nicht auf komplexen Protokollen oder Datenformaten aufgebaut werden.

Ein oft verwendetes Zitat stammt von Ian Robinson: *Be of the web, not behind the web.* Es sollen möglichst offene, weit verbreitete Standards aus dem Internet verwendet werden. HTTP, REST und JSON sind Beispiele hierfür.

Einsatz passender Technologien

Die Umsetzung einer Anwendung in Form eines Monolithen mit zentral vorgegebener Architektur erfordert auch, dass sämtliche Bestandteile mit demselben Technologie-Stack umgesetzt werden. Durch die Aufteilung eines Monolithen in Microservices entfällt dieser Zwang, und Sie können für jeden Service entscheiden, welches Werkzeug oder welcher Technologie-Stack für die Aufgabe am besten geeignet ist, und dieses bzw. diesen entsprechend einsetzen. Durch die klaren Schnittstellen über standardisierte Protokolle bleiben die Services interoperabel, d. h. sie können weiterhin zusammenarbeiten.

Der Rahmen, der von einer zentralen Architektur vorgegeben wird, kann z. B. allgemeine Vorgehensweisen zu Security, Monitoring oder auch Logging bestimmen, die die Schnittstellen erfüllen müssen. Die Umsetzung ist dann dem einzelnen Service überlassen. Eine Vorgabe könnte z. B. darin bestehen, dass die Services mit *OAuth* abgesichert oder Log-Statements in einem bestimmten Format in ein zentrales Log Management Tool abgelegt werden müssen.

Sidecar Pattern

In modernen Microservice-Infrastrukturen können Funktionalitäten, die von mehreren Services benötigt werden, als separate Komponenten oder Services zur Verfügung gestellt werden. Funktionalität wie z. B. Konfiguration, Sicherheitsprüfungen oder die Verschlüsselung von Netzwerkverbindungen werden nicht mehr in jedem einzelnen Service implementiert, sondern nur noch mit dem jeweiligen Service verknüpft. Diese als Beiwagen oder Sidecar bzw. Sidekick bezeichneten Komponenten

oder Services werden nicht als Teil des Anwendungsquellcodes implementiert und können technologisch davon unabhängig sein. Die Infrastruktur stellt sie zur Verfügung, und die einzelnen Services können Basisfunktionalität an die Infrastruktur delegieren.

Dezentrale Datenhaltung

Wie bereits bei den *Bounded Contexts* erwähnt, besitzt jeder Microservice ein eigenes Domänenmodell und somit Daten, für die er verantwortlich ist. Im Gegensatz zu Monolithen, die oftmals ihre Daten in einer großen, zentralen relationalen Datenbank ablegen, entscheidet jeder Service, wie seine Daten persistiert werden sollen. Für manche Anwendungsfälle ist eine Speicherung in einer relationalen Datenbank sinnvoll, in anderen Fällen können No-SQL-Datenbanken von Vorteil sein. Die relationale Datenbank kann zentral zur Verfügung gestellt werden, allerdings sollten die von den Services genutzten Schemata nicht in der Datenbank verknüpft sein.

Auftretende Redundanzen in den Domänenmodellen und somit bei der Speicherung der Daten über mehrere Services hinweg werden in Kauf genommen und minimieren die Kohäsion zwischen den Services.

Werden die Daten nicht mehr von einer monolithischen Anwendung mit einem zentralen Transaktionsmanager in einer Datenbank gespeichert, ergeben sich daraus Konsequenzen, mit denen Sie sich bei Microservices beschäftigen müssen.

Innerhalb eines Monolithen ist es z. B. einfacher, die Konsistenz des Gesamtsystems bei der Arbeit über mehrere Komponenten hinweg mittels Transaktionen sicherzustellen. Bei einzelnen, verteilten Services würde dies zwangsweise zu verteilten Transaktionen führen, die zum einen nur mit viel Aufwand umzusetzen sind und zum anderen die Services wieder enger miteinander koppeln würden.

Klassische Transaktionen und somit eine strikte Konsistenz der Datensätze finden bei Microservices nur innerhalb eines Service statt. Bei Operationen, die sich über mehrere Servicegrenzen hinweg erstrecken, muss von einer *Eventual Consistency* ausgegangen werden. Die Konsistenz von Datensätzen stellt sich, sofern kein weiterer Schreibvorgang durchgeführt wird bzw. kein Fehler auftritt, erst in der Zukunft mit Sicherheit ein. Im Fehlerfall müssen Sie mit Kompensationslogik und eventuell Aufrufen sicherstellen, dass der ursprüngliche Zustand der Daten wiederhergestellt wird.

Der Umgang mit Inkonsistenzen, also Widersprüchlichkeiten innerhalb der Daten, ist bei Microservices elementar und bildet oftmals auch die Situation im »realen Leben« ab. Die Genehmigung eines Schadensfalls bei einer Versicherung führt z. B. meist auch nicht zu einer sofortigen Gutschrift auf das Konto des Geschädigten, sondern es vergeht noch etwas Zeit, bis das Geld verbucht ist.

Im Versicherungsbeispiel ist die Bearbeitung des Schadensfalls in dem einem System bereits abgeschlossen, allerdings wartet der Kunde noch auf sein Geld, da die Buchung in einem anderen System noch nicht durchgeführt wurde.

Dezentrale Verwaltung

Für viele Microservice-Befürworter ist die Abgrenzung zu service-orientierten Architekturen (*SOA*) wichtig und elementar. Es ist allerdings sehr schwer, eine harte Abgrenzung vorzunehmen, da auch der Begriff SOA für viele Unterschiedliches bedeutet, da es, ebenso wie bei den Microservices, keine klare, einheitliche Definition gibt.

Für SOA-Befürworter sind Microservices genau das, was sie bisher schon gemacht haben, anderen hingegen fehlt die zentrale Verwaltung und Steuerung z. B. in Form eines Enterprise Service Bus (*ESB*).

Die zweite Auslegung, also das Fehlen einer zentralen Verwaltungsinstanz, ist sicher die verbreitetere und führt bei zahlreichen Microservice-Befürwortern zur strikten Ablehnung einer SOA. Microservices besitzen keine zentrale Kontrolleinheit, und auch ihr Nachrichtenfluss wird nicht zentral gesteuert. Jeder Service entscheidet für sich, wohin welche Nachricht geroutet wird.

Ziel ist die maximale Entkopplung der einzelnen Services voneinander und nicht die Einführung einer zentralen Verwaltungsinstanz. Die Architektur der Servicelandschaft kann mit derjenigen des Internets verglichen werden.

Ein Service von nur einem Team

Eine Frage, die sich bei der Entwicklung von Microservices häufig stellt, besteht darin, wie groß ein Service sein soll und wie groß das Team sein soll, das diesen Service entwickelt. Auch hierfür gibt es keine Regeln.

Die Größe des Microservice sollte sich immer an der umzusetzenden Geschäftsfunktionalität orientieren. Die Größenspanne eines Service reicht von einem *Function as a Service*, dem *Serverless Computing*, bei dem, einfach ausgedrückt, jede Funktion als separater Service zur Verfügung gestellt wird, bis zu größeren, an Monolithen erinnernde Anwendungen. Eine von einigen Teams aufgestellte Definition, dass jeder Microservice genau ein REST-Interface besitzt, spiegelt nicht die Idee von Microservices wider. Services können, soweit sie fachlich zusammengehören und gemeinsam in einem *Bounded Context* definiert sind, mehrere REST-Interfaces anbieten.

In vielen Projekten hat sich die Regel als sinnvoll herausgestellt, dass ein Service von genau einem Team erstellt und gepflegt wird. Das Team muss in diesem Fall crossfunktional aufgestellt, d. h. aus entsprechenden Spezialisten zusammengestellt sein. Der komplette Service muss von diesem Team allein verwaltet werden können.

Diese »ein Service – ein Team«-Regel gilt jedoch nicht umgekehrt: Ein Team kann sich um mehrere fachlich zusammenhängenden Services kümmern.

Wie bereits bei *Bounded Context* und *Conways Law* erwähnt, sollen mit den crossfunktionalen Teams Kommunikations-Overhead und Interessenskonflikte zwischen den einzelnen technischen Teams minimiert werden.

Jedes Team soll die Funktionalität seines Service autark umsetzen können, d. h. möglichst wenig oder gar nicht abhängig von anderen Teams sein.

Ideale Teamgröße für Microservices

Über die ideale Teamgröße gibt es ebenfalls die verschiedensten Aussagen. Laut Martin Fowler gibt es bei Amazon z. B. die »Zwei-Pizza-Team«-Größe, was so viel bedeuten soll, dass zwei Pizzen ausreichen müssen, damit das Team satt wird. Über die Größen der Pizzen kann spekuliert werden.

In der Realität haben sich Teamgrößen von drei bis maximal 12 Personen pro Service als sinnvoll herauskristallisiert. Bei mehr als 12 Personen nimmt der Kommunikations-Overhead wieder zu, und bei weniger als drei Personen darf eventuell nicht mehr von einem Team gesprochen werden.

Infrastruktur und Automatisierungen

Die Geschwindigkeit, mit der Teams neue Versionen ihrer Software veröffentlichen können, trägt nach Untersuchungen und Befragungen von Unternehmen zum Erfolg eines Projekts bei. Nicole Forsgren, Jez Humble und Gene Kim beispielsweise haben über Umfrageergebnisse festgestellt, dass es einen Zusammenhang zwischen dem Tempo, in dem ein Team neue Releases veröffentlichen kann, und seiner über verschiedene Indikatoren gemessenen Leistung gibt. Solche Indikatoren sind z. B. die Fehleranzahl in der Software oder die Geschwindigkeit, in der neue Features umgesetzt werden können. In der Summe wird das wichtigste Ziel überhaupt erreicht: Die Zufriedenheit der Kunden steigt.

Durch den Einsatz und die Prinzipien von *Continuous Integration* und *Continuous Delivery* können Sie, neben der Qualitätssicherung, in Ihren Projekten die Geschwindigkeit bei der Erstellung und Auslieferung von Softwarereleases erhöhen und somit positiv zum Projekterfolg beitragen.

Die Lösungen für ein automatisiertes Deployment haben sich in den letzten Jahren von skriptbasierten Ansätzen zu komplett containerbasierten Varianten weiterentwickelt. Ein Deployment einer Software bedeutet bei den meisten Microservices nur noch das Starten eines zuvor erstellten Containers, der bereits komplett getestet wurde.

Automatisierte Tests sind bei der Erstellung von Microservices essenziell. Nur so können Sie bei Änderungen schnell und sicher feststellen, ob der Funktionsumfang der Anwendung noch vollständig ist.

Die Größe der Services ist für die Geschwindigkeit der Testausführung und des Deployments ein maßgebender Faktor. Je weniger Funktionalität im Service steckt, desto weniger Tests müssen ausgeführt werden, und je weniger Code enthalten ist, umso kleiner wird auch die auszuliefernde Software. Lassen Sie sich bei Entscheidungen zur Aufteilung von Services oder für die Entwicklung der Software nicht von technischen Rahmenbedingungen leiten. Behalten Sie immer den *Bounded Context* des Service im Hintergrund und definieren Sie die Servicegrenzen anhand der abzubildenden Funktionalität.

Wie bereits erwähnt, nutzen viele microservice-basierten Projekte Containertechnologien, um ihre Software bereitzustellen. Dadurch wird die Anwendungssoftware von der ausführenden Hardware entkoppelt und erhöht die Flexibilität beim Einsatz. Sie können Container erstellen, kopieren, ausführen und zwischen verschiedenen Umgebungen verschieben. Somit können Sie einen Container, den Sie in einer Entwicklungsumgebung erstellt und getestet haben, bis auf eine Produktionsumgebung verschieben und dort ausführen.

Lastspitzen in Ihrer Anwendung können Sie durch den Start einer zusätzlichen Instanz in Form eines Containers abfedern. Skaliert werden muss lediglich der Anwendungsteil, der tatsächlich benötigt wird. Bei Deployment-Monolithen ist es oftmals trotz Automatisierung schwierig, eine weitere Instanz auszuführen, da eventuell ein neuer Server oder eine komplette Betriebssysteminstanz benötigt wird.

Die schnelle Bereitstellung von neuen Servern bzw. neuen Containern und somit die Skalierung der Services kann zu einem Erfolgsfaktor einer Anwendung werden. Viele Firmen nutzen aus diesem Grund Cloud-Plattformen von großen Anbietern, die eine unbegrenzte Skalierbarkeit versprechen und innerhalb sehr kurzer Zeit neue Instanzen zur Verfügung stellen können.

Auf Fehlersituationen vorbereitet

Die Zerlegung einer Anwendung in viele kleinere Einzelteile erfordert die Betrachtung von möglichen Fehlersituationen bei der Kommunikation zwischen den einzelnen Services. Durch z. B. Deployments oder technische Fehler ist ein Service potenziell nicht immer nutzbar.

Deployment-Monolithen haben hier den Vorteil, dass sämtliche Anwendungsbestandteile mitausgeliefert werden und eine solche Situation nicht auftreten kann. Fehlende Bestandteile oder Inkompatibilitäten zwischen Komponenten würden bei Tests schon viel früher auftreten und nicht erst zur Laufzeit.

In Microservice-Umgebungen spielt die Überwachung der einzelnen Services und der zugehörigen Infrastruktur eine zentrale Rolle. Mit ihrer Hilfe können Fehler frühzeitig erkannt und teilweise direkt und automatisiert behoben werden. Für die Analyse der Fehlersituationen sollten Sie zusätzlich auf ein aussagekräftiges und umfassendes Logging und Tracing achten. Zentrale Logging-Lösungen führen z. B. sämtliche Logs von Anwendungen und Infrastrukturkomponenten an einer Stelle zusammen, wo sie dann ausgewertet werden können. Jede Komponente sendet hierbei ihre Daten an diese zentrale Instanz, anstatt sie lokal zu halten. Werden neue Service-Instanzen gestartet, können diese direkt ihre Log-Daten schicken und müssen nicht mehr einzeln für einen Datenabruf konfiguriert werden.

Für das Monitoring der Anwendung ist es sinnvoll, sich bereits bei der Implementierung Gedanken über mögliche Metriken zu machen. Diese können entweder rein technischer Natur oder auch fachlich motiviert sein. Beim Über- oder Unterschreiten eines Schwellwertes kann ein Alarm ausgelöst werden. Werden z. B. in einem bestimmten Zeitraum zu viele nicht-HTTP-OK (200)-Status-Codes zurückgeliefert, könnte dies bereits auf ein Problem hindeuten. Bereits der Rückgang von beispielsweise Buchungszahlen kann ein Indiz dafür sein, dass in der Anwendung Probleme auftreten, beispielsweise, dass eine Oberfläche so angepasst wurde, dass die Nutzer nun Probleme damit haben, eine neue Buchung durchzuführen. Es muss nicht immer ein Infrastrukturproblem vorliegen.

Um die Verfügbarkeit ihrer kompletten Anwendung auch beim Ausfall einzelner Bestandteile zu prüfen bzw. sicherzustellen, sind einige Firmen dazu übergegangen, mit einem sogenannten *Chaos-Monkey* im Testbetrieb zufällig ausgewählte, einzelne Microservice-Instanzen zu stoppen und darüber Ausfälle zu simulieren. Dieses Vorgehen kann dabei helfen, Verfügbarkeitsprobleme frühzeitig zu erkennen und diese umgehend zu adressieren, statt erst im Betrieb darauf zu stoßen. Sie sollten immer daran denken, dass jeder Service zu jeder beliebigen Zeit einmal nicht verfügbar sein könnte, und dementsprechende Vorkehrungen treffen. Fehlertoleranz ist kein Feature, sondern eine Anforderung.

Beim Einsatz einer synchronen Kommunikation zwischen verschiedenen Services wie z. B. mit HTTP-REST-Calls stehen Ihnen verschiedene Möglichkeiten zur Verfügung, um die höhere Fehlertoleranz Ihres Clients zu verbessern:

- Setzen eines entsprechenden Network-Timeouts
- Konfiguration von *Retry-Strategien* für Aufrufe
- Einführung von *Semaphoren* oder Thread-Pools für Backend-Aufrufe
- Nutzung eines *Circuit Breakers*

Die Einführung von *Semaphoren* oder Thread-Pools für Backend-Aufrufe ermöglicht es Ihnen, die Anzahl der Aufrufe zu einem Backend zu kontrollieren und im Bedarfsfall zu limitieren. Falls zu viele Aufrufe nicht oder fehlerhaft beantwortet werden, können Sie darauf reagieren und z. B. die Kommunikation zum Service für einige Zeit einstellen, bevor Sie es wieder probieren. Letzteres wird als *Circuit Breaker* bezeichnet, da der Mechanismus ähnlich einem Schutzschalter verhindern soll, dass fehlerhafte oder langsame Remote-Calls das Komplettsystem in Mitleidenschaft ziehen.

Besitzt Ihre Anwendung eine Anbindung an mehrere Backend-Systeme, bietet die Entkopplung der eingehenden und ausgehenden Aufrufe zusätzlich den Vorteil, dass ein fehlerhaftes oder langsames Backend nicht dazu führt, dass Ihr kompletter Service blockiert. Würden eingehende Aufrufe ohne Semaphore oder Thread-Pools direkt ein Backend aufrufen und würde dieses sehr langsam reagieren, müsste der eingehende Aufruf so lange warten, bis die Backend-Anfrage beantwortet ist. Jeder neue eingehende Call, der dieses Backend intern aufruft, würde wieder blockieren. Dies würde sich so lange fortsetzen, bis schließlich alle Threads, die eingehende Anfragen beantworten, in Verwendung sind. Eingehende Aufrufe, die dieses Backend gar nicht nutzen, könnten ebenfalls nicht mehr verarbeitet werden. Der Einsatz eines Thread-Pools hingegen kann die Anzahl der Backendaufrufe limitieren und neue eingehende Aufrufe direkt behandeln bzw. abweisen. Somit bleiben Threads für eingehende Aufrufe verfügbar, und Ihr Service kann weiterhin Anfragen bearbeiten.

Alternativ zu synchronen Remote-Calls setzen viele Microservice-Umgebungen auf eine asynchrone Kommunikation, um Services weiter zu entkoppeln. Entsprechende Integration Patterns, wie z. B. die Umsetzung einer *Event Driven Architecture*, verhelfen Ihrer Anwendung hierbei unter anderem zu besserer Skalierbarkeit und höherer Ausfallsicherheit.

Ist ein Service nicht verfügbar, kann eine Nachricht zwischengespeichert und später ausgeliefert werden. Auch die Antwortzeit des gerufenen Service ist nicht mehr entscheidend für die Weiterverarbeitung des Clients und kann in der Folge auch nicht zu Timeout-Problemen führen.

Da eine Microservice-Architektur meist aus vielen Services besteht, wird jeder Service wiederum andere Services aufrufen müssen. Setzen Sie auf eine synchrone Kommunikation, ergibt sich die Laufzeit eines Service Requests mindestens aus der Summe der Laufzeit aller aufgerufenen Services. Bei einer asynchronen Kommunikation müssen nur die Nachrichten ausgeliefert werden, und die Laufzeit ist dementsprechend kürzer.

> *Almost all the successful microservice stories have started with a monolith that got too big and was broken up. – Martin Fowler*

2.3 Module und Libraries

Eines der wichtigsten Konzepte in der Softwareentwicklung ist die Modularisierung. Sie ermöglicht uns, große komplexe Probleme in kleinere, beherrschbare Probleme zu zerlegen und damit auch die Flexibilität und Wartbarkeit der Software zu erhöhen.

In Abschnitt 2.1.8, »Der Go-Workspace – Wie organisiere ich Go-Code?«, haben wir bereits die Begriffe *Package*, *Import Path*, *Module* und *Repository* kennengelernt. Packages, Module und Repositories stellen für Go die Einheiten zur Modularisierung dar.

In einem Repository können sich ein oder mehrere Module befinden. Ein Modul wiederum kann ein oder mehrere Packages enthalten. Jedes Package beinhaltet dann die einzelnen Go-Quellcode-Dateien.

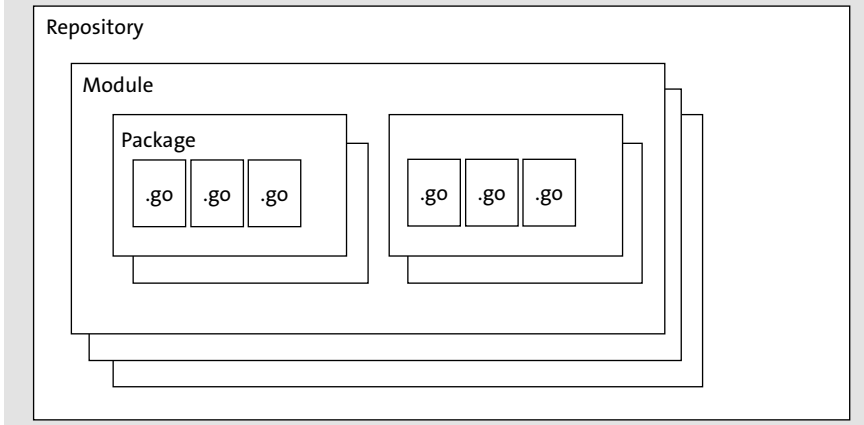

Abbildung 2.19 Struktur von Go-Sourcecode

Abhängigkeiten zwischen Packages werden immer über die Angabe eines Import Path im Kopfteil einer Source-Datei angegeben.

```
import (
   "github.com/sirupsen/logrus"
)
```

Listing 2.85 Definition einer Abhängigkeit zu einem anderen Package

Der angegebene Import Path bezieht sich auf ein anderes Package, das sich zum Kompilierungszeitpunkt lokal auf Ihrem Computer befinden muss. Wie Sie den Download über die Go-Umgebung anstoßen bzw. durchführen können, bekommen Sie in Abschnitt 2.3.1, »Third-Party Libraries einbinden«, gezeigt.

Wo die Abhängigkeiten bei Ihnen lokal gesucht werden, hängt von Ihrer Go-Version und Ihrem Projekt ab. Mehrere Ablageorte kommen in Betracht:

- Direkt im *src* Verzeichnis des Go-Workspaces
- *Vendor* Verzeichnis innerhalb des jeweiligen Projekts
- Unterhalb des *mod*-Verzeichnisses des Go-Workspaces

Die von der Code-Organisation her einfachste Variante ist gegeben, wenn sich der gesamte Code innerhalb des *src*-Verzeichnisses des Go-Workspace befindet und alle Abhängigkeiten dort aufgelöst werden können. Dies war auch der ursprüngliche Ansatz von Go. Ein *Dependency-Management*-Mechanismus war nicht vorhanden.

Allerdings hat dies den großen Nachteil, dass eine Abhängigkeit nur in einer Version im Go-Workspace vorhanden sein kann. Alle Projekte innerhalb des Workspace müssen dieselbe Version dieser Abhängigkeit verwenden. Benötigen mehrerer Ihrer Projekte dieselbe Bibliothek in verschiedenen Versionen, können Sie dies mit dem hier beschriebenen Ansatz nicht einfach umsetzen. In der Folge haben sich verschiedene Lösungsansätze verbreitet:

- Kopieren aller Abhängigkeiten in das Projekt selbst und Umschreiben sämtlicher Import-Statements
- Kopieren der Abhängigkeiten in das Projekt und Anpassen des GOPATH, sodass diese Packages für das entsprechende Projekt benutzt werden
- Festhalten einer Abhängigkeitsversion in einer Datei und Durchführung eines entsprechenden Updates auf die passende Version, bevor mit dem Projekt gearbeitet wird
- Komplettes Duplizieren des Go-Workspace für jedes Projekt

Auch diese Ansätze bergen jedoch jeweils ihre Probleme. Entweder, Sie haben einen großen Aufwand, sämtliche Abhängigkeiten durch Umschreiben der Import-Statements anzupassen, oder Sie benötigen ein zusätzliches Build-Skript, das Ihnen die Umgebung vor der Arbeit mit dem Projekt richtig anpasst. Beides widerspricht den Grundideen von Go!

In der Go-Version 1.6 (in Version 1.5 noch experimentell) wurden die sogenannten *vendor*-Verzeichnisse eingeführt, die diese Problemstellung aufgreifen. Innerhalb jedes Projekts können Sie seitdem ein Verzeichnis mit dem Namen *vendor* anlegen, in dem sich alle für das Projekt nötigen externen unangepassten Abhängigkeiten ablegen lassen. Die Pfade innerhalb des Verzeichnisses entsprechen analog dem *src*-Verzeichnis dem Import Path des Projekts.

Wie in Abbildung 2.20 dargestellt, wurde im Projekt `first-vendor`, das sich im *src*-Verzeichnis des Go-Workspace befindet, die Abhängigkeit `github.com/sirupsen/logrus` in das Vendor-Verzeichnis aufgenommen.

Abbildung 2.20 Vendor-Verzeichnis in eigenem Projekt

Andere Projekte können nicht auf Abhängigkeiten innerhalb eines *vendor*-Verzeichnisses zugreifen, und bei der Kompilierung wird zum Auflösen von Abhängigkeiten zuerst das *vendor*-Verzeichnis verwendet.

Vendor-Verzeichnisse

Vendor-Verzeichnisse werden nur bei nicht-Go-Modules-basierten Projekten standardmäßig eingebunden. Bei Go-Modules-Projekten muss das Flag `-mod=vendor` mitgegeben werden.

Mit der Einführung von Go Modules erhielt Go einen Standard-Dependency-Management-Mechanismus, der in Version 1.14 als »production ready« gekennzeichnet wurde. Diesen sollten Sie für neue Projekte auf jeden Fall verwenden.

Go Modules verwenden!

Verwenden Sie in neuen Projekten für die Verwaltung Ihrer Abhängigkeiten Go Modules.

Die Ablage der Abhängigkeiten erfolgt weiterhin im Go-Workspace, allerdings nicht mehr direkt im *src*-Verzeichnis oder dem *vendor*-Verzeichnis im Projekt, sondern in einem separaten *mod*-Verzeichnis unterhalb des *pkg*-Verzeichnisses, wie in Abbildung 2.21 gezeigt. Welche Abhängigkeiten in welcher Version verwendet werden sollen, gibt eine zusätzliche *go.mod*-Datei im Projekt an. Jedes Go-Modules-basierte Projekt muss eine solche Datei enthalten. Mehr dazu erfahren Sie in Abschnitt 2.3.3, »Module erstellen, verwenden und versionieren«.

Die Abhängigkeiten werden, im Unterschied zum *src*- oder *vendor*-Verzeichnis, versioniert abgelegt. Wenn Sie beispielsweise die Log-Bibliothek Logrus in einem Go-Modules-basierten Projekt verwenden, wird von der Go-Umgebung hinter dem letzten Pfadbestandteil des Import Path ein @, gefolgt von der jeweiligen Versionsnummer, angehängt. Es ergibt sich dann etwa der Pfad *github.com/sirupsen/logrus@1.4.2*. Wie

in Abbildung 2.22 dargestellt, können Sie somit mehrere Versionen der gleichen Bibliothek lokal speichern.

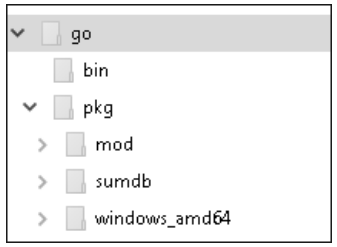

Abbildung 2.21 Mod-Verzeichnis unterhalb des pkg-Verzeichnisses

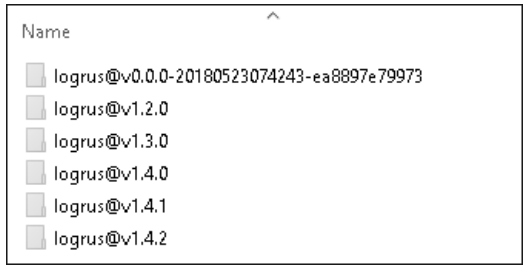

Abbildung 2.22 Beispiel eines mod-Verzeichnisses einer Bibliothek

2.3.1 Third-Party Libraries einbinden

Wie bereits beschrieben, müssen Sie Abhängigkeiten zwischen Packages immer mit einem Import-Statement innerhalb einer Go-Quellcodedatei angeben. Zum Kompilierungszeitpunkt muss sich das eingebundene Package lokal bei Ihnen auf der Festplatte befinden. Den entsprechenden Download der Abhängigkeit übernimmt die Go-Umgebung für Sie. Hierzu haben wir bereits das Tool `go get` kennengelernt.

Der erste Schritt, nachdem Sie sich ein neues Projekt auf die Festplatte kopiert haben, ist immer der Aufruf von `go get` im Projekt-Root-Verzeichnis. Damit laden Sie alle benötigten Abhängigkeiten herunter und können das Projekt kompilieren.

Was wird bei go get heruntergeladen?

Wenn Sie mit `go get` eine Abhängigkeit installieren möchten, geben Sie als Parameter einen Import Path an. Heruntergeladen wird aber nicht nur dieses eine Package, sondern das komplette Repository der Abhängigkeit. Für den Download stellt dieses Repository die kleinste Einheit dar. Wenn Sie z. B. nur das Package `github.com/sirupsen/logrus/hooks/syslog` angeben, wird trotzdem das Repository `github.com/sirupsen/logrus` mit allen enthaltenen Packages heruntergeladen.

Möchten Sie in eine Quellcode-Datei eine neue Abhängigkeit aufnehmen, gehen Sie wie folgt vor:

- Aufruf von `go get` mit entsprechendem Import Path als Parameter
- Aufnahme des Import-Statements und Nutzung der Bibliothek
- Kompilierung der Anwendung

Gofmt und neue Imports

In IDEs mit eingeschalteter Autoformatierung und Import-Bereinigung kann es unter Umständen störende Seiteneffekte haben, wenn Sie ein neues Import-Statement aufnehmen, dies aber noch nicht im Quellcode verwenden. Beim Speichern wird dann der Import direkt wieder gelöscht (Sie verwenden den Import ja noch nicht). Wenn Sie, wie der Autor, ständig Ihre Änderungen speichern, kann es sinnvoll sein, zuerst die Bibliothek zu verwenden und dann ein Import-Statement dafür aufzunehmen, oder eben an dieser Stelle auf die Gewohnheit des ständigen Speicherns zu verzichten.

Das Vorgehen zur Aufnahme einer externen Bibliothek wollen wir uns anhand eines kleinen Beispiels ansehen. Verwendet wird die bereits mehrfach genannte Log-Bibliothek Logrus. Ihre Schnittstelle ist weitgehend kompatibel zum `log`-Package aus dem Go-Standardumfang, sie bietet allerdings einen deutlich größeren Funktionsumfang und kann besser konfiguriert werden.

Starten wir mit einer »Hello World«-Anwendung, die das Log-Package aus der Standardbibliothek verwendet. Legen Sie die Datei *main.go* im Verzeichnis *$GOPATH/src/first-dependency* ab.

```
package main

import "log"

func main() {
    log.Println("Hello World")
}
```

Listing 2.86 Beispielanwendung mit Log-Package aus Standardbibliothek

Durch einen `go get`-Aufruf installieren Sie die Bibliothek lokal auf Ihrem Computer, damit sie später beim Kompilieren gefunden wird:

```
go get github.com/sirupsen/logrus
```

Innerhalb der Anwendung müssen Sie ein entsprechendes Import-Statement aufnehmen, um Logrus verwenden zu können. Die Funktion `Println` ist im Package `github.com/sirupsen/logrus` enthalten, und das entsprechende Import-Statement muss folglich so aussehen:

```
import "github.com/sirupsen/logrus"
```

Listing 2.87 Importstatement für Einbindung von Logrus

Da die Logrus-Bibliothek weitgehend kompatibel zur Standard-`log`-Implementierung ist und wir den eigentlichen Anwendungscode nicht anpassen möchten, können wir über einen Package Alias das `logrus`-Package unter dem Namen `log` referenzieren. Das Import-Statement sieht dann im kompletten Code des Beispiels so aus:

```
package main

import log "github.com/sirupsen/logrus"

func main() {
    log.Println("Hello World")
}
```

Listing 2.88 Beispielanwendung mit Logrus

Über `go build main.go` bzw. `go run main.go` können Sie nun die Anwendung kompilieren bzw. ausführen.

Im obigen Beispiel haben wir der Einfachheit halber ein Legacy-Projekt, das sich direkt im Go-Workspace befindet, verwendet. Die Einbindung einer externen Bibliothek in ein Go-Modules-basiertes Projekt unterscheidet sich hiervon nicht wesentlich. In Abschnitt 2.3.3, »Module erstellen, verwenden und versionieren«, sehen wir uns Go Modules noch einmal an. An dieser Stelle sei nur erwähnt, dass der Aufruf von `go get` bei Go-Modules-basierten Projekten auch die *go.mod*-Datei, die sämtliche Abhängigkeiten mit Versionsangabe auflistet, entsprechend aktualisiert.

Versionsangaben bei go get

Beim Einsatz von Go Modules haben Sie zusätzlich die Option, beim `go get`-Aufruf eine Versionsangabe mitzugeben. Diese Versionsangabe wird in die *go.mod*-Datei übernommen, und die entsprechende Version der Bibliothek wird verwendet.

Der Aufruf sieht dann z. B. wie folgt aus:

```
go get github.com/sirupsen/logrus@1.4.2
```

Bei Legacy-Projekten ist bei der Installation einer Abhängigkeit die Angabe einer Version nicht möglich, da diese zentral im Workspace abgelegt wird. Es wird immer automatisch die neuste Version verwendet!

Manche Bibliotheken müssen in den eigenen Sourcecode importiert, aber nicht direkt verwendet werden. Das ist z. B. bei SQL-Bibliotheken der Fall. Hier werden beim Einbinden der Bibliothek Initialisierungen vorgenommen, so dass eine Verbindung

zur Datenbank aufgebaut werden kann. In diesem Fall müssen Sie den *Blank Identifier* beim Import verwenden, damit der entsprechende Import nicht automatisch entfernt oder als Fehler ausgewiesen wird.

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

Listing 2.89 Einbinden einer SQL-Bibliothek über Blank Identifier

2.3.2 Eigene Libraries erstellen und teilen

In den vorangegangenen Kapiteln haben wir immer externe Bibliotheken in unsere Anwendungen aufgenommen. Möchten Sie selbst eine Bibliothek zur Verfügung stellen, müssen Sie sich dazu vorab ein paar Gedanken machen:

- Wie soll die Bibliothek ausgeliefert werden?
- Welchen Packagenamen bzw. welchen Import Path soll die Bibliothek besitzen?
- Welche Bestandteile sollen für Anwender der Bibliothek nutzbar sein?

Import Compatibility Rule

Da Go sehr großen Wert auf Kompatibilität zwischen verschiedenen Versionen legt, sollten Sie immer die vom Go-Team vorgeschlagene »Import compatibility rule« beherzigen:

»If an old package and a new package have the same import path, the new package must be backwards compatible with the old package.«

Die beiden ersten der oben dargestellten drei Fragen hängen sehr eng miteinander zusammen. Da in Go bereits implizite Logiken zum Download von Packages eingebaut sind, ist der Name des entstehenden Packages bzw. des entsprechenden Import Path nicht völlig frei wählbar. Je nach Package-Name wird die Go-Umgebung versuchen, das Package von verschiedenen Orten herunterzuladen (siehe hierzu auch »get – Abhängigkeiten laden und installieren« in Abschnitt 2.1.9.

Den Package-Namen müssen Sie immer vollqualifiziert definieren, so dass die Umgebung bei einem `go get`-Aufruf die Abhängigkeit auch herunterladen kann. Befindet sich Ihr Quellcode nicht bei einem der großen, vorkonfigurierten Anbieter wie GitHub.com, müssen Sie unter Umständen im Import-Statement noch ein Download-Protokoll angeben. Für ein direkt über SSH angesprochenes Git-Repository sieht dies z. B. so aus (Anhängen von `.git` an den Package-Namen):

```
go get myserver.local/volume2/git-server/somelib.git
```

Das entsprechende Import-Statement sieht in diesem Fall wie folgt aus (bei der Verwendung des Packages entfällt die Endung `.git`.):

```
import (
   "myserver.local/volume2/git-server/somelib.git"
)
…
somelib.MeineFunktion
```

Listing 2.90 Import-Statement und Verwendung einer Abhängigkeit, die über SSH ausgeliefert wird

Seit der Einführung des neuen Dependency-Mechanismus »Go Modules« werden Bibliotheken standardmäßig über HTTP heruntergeladen, wodurch auch eine Entkopplung zwischen Repository und URL erreicht wird bzw. werden kann. Ihr Source Repository muss sich nicht mehr genau unter der im Import Path angegebenen vollqualifizierten URL befinden. Ein Web-Server muss lediglich unter der angegebenen URL dem Client mittels HTML-Meta-Tag die genaue Adresse und den Typ des Repositorys mitteilen.

Befindet sich Ihr Repository aktuell z. B. bei GitHub.com, und Sie möchten einen alternativen Import Path wie `golang.source-fellows.com/cmd` verwenden, können Sie das Meta-Tag `go-import` innerhalb der unter *https://golang.source-fellows.com/cmd?go-get=1* zurückgelieferten HTML-Seite verwenden. Sie erkennen die entsprechende URL des Import Paths wieder. Der Query-Parameter `go-get` ermöglicht es Web-Servern, das Meta-Tag nicht immer mit ausliefern zu müssen und eventuell Auswertungen über Nutzung der Bibliothek zu erstellen. Wird der Parameter nicht angegeben, muss der Server das Meta-Tag nicht liefern.

```
<meta name="go-import" content="golang.source-fellows.com git https:// ↩
github.com/kkoehler/golang">
```

Listing 2.91 Beispiel für das go-import-Meta-Tag

Im Beispiel kann man nun über den Import Path `golang.source-fellows.com/cmd` das eigentliche Repository unter *https://github.com/kkoehler/golang* referenzieren:

```
import (
   "golang.source-fellows.com/cmd"
)
```

Go-Import-Protokoll

Tools wie etwa Gitlab haben das Go-Import-Protokoll bereits umgesetzt. Abfragen der Go-Umgebung auf ein Repository innerhalb eines Gitlab-Servers liefern direkt ein passendes `go-import`-Meta-Tag.

An dieser Stelle können Sie ohne weitere Schritte die Repository-URL des Servers als Import Path verwenden. Auch auf eine Angabe des Repository-Typs können Sie hierbei verzichten.

Läuft Ihr Gitlab-Server z. B. unter *https://src.source-fellows.com* und Ihr Repository liegt dort unter */training/samples*, so können Sie diesen Pfad direkt verwenden. Der resultierende Import Path lautet:

```
src.source-fellows.com/training/samples
```

Achten Sie bitte auch darauf, ob ein Module Proxy und die Sum-DB verwendet werden sollen (siehe hierzu die Abschnitte zu GOPROXY, GONOPROXY, GOPRIVATE bzw. GOSUMDB, GONOSUMDB in Abschnitt 2.1.7). Im obigen Beispiel können Sie z. B. das Präfix des Import Path golang.source-fellows.com in GOPRIVATE aufnehmen:

```
go env -v GOPRIVATE=golang.source-fellows.com
```

Sobald Sie sich für einen passenden Package-Namen und somit für eine Auslieferungsvariante entschieden haben, können Sie ihre Bibliothek veröffentlichen. Welche Bestandteile Ihres Packages für Clients sichtbar sein sollen, steuern Sie über den normalen Export-Mechanismus. Hier unterscheidet sich die Entwicklung nicht von einer Anwendung.

In den meisten bisherigen Beispielen haben wir im Main-Package gearbeitet. Für eine Bibliothek müssen Sie einen passenden Package-Namen verwenden, der im Client eingebunden werden kann. Die `somelib` aus dem obigen SSH/Git-Beispiel könnte dementsprechend wie in Listing 2.92 dargestellt aussehen:

```
package somelib

import "fmt"

// CallLib calls something.
func CallLib() {
   fmt.Println("Here is somelib")
}
```

Listing 2.92 Beispiel-Bibliothek im Package somelib

Durch das Einchecken der Bibliothek in eine Versionsverwaltung steht Ihre Bibliothek anderen Anwendern zur Verfügung, diese können über den entsprechenden Import Path auf Ihre Bibliothek zugreifen.

Haben Sie das oben genannte Beispiel zum Beispiel bei GitHub unter *https://github.com/SourceFellows/somelib* eingecheckt, so können Sie diese Abhängigkeit mit dem Kommando `go get github.com/sourcefellows/somelib` lokal installieren und nutzen. Ein möglicher Client ist in Listing 2.93 dargestellt.

```
package main

import "github.com/sourcefellows/somelib"

func main() {
    somelib.CallLib()
}
```

Listing 2.93 Verwendung einer eigenen Bibliothek aus einem Git-Repository

Wie bereits weiter oben erwähnt, ist die kleinste auszuliefernde Einheit Ihr komplettes Repository. Das kann Vor-, aber auch Nachteile haben, je nachdem, welchen Verwaltungsansatz Sie mit Ihrem Repository verfolgen.

Bei Mono-Repositories kann z. B. die Verwendung eines einzelnen Packages daraus einen recht umfangreichen Download nach sich ziehen. Auf der anderen Seite sollte in diesem Fall ohnehin Ihr kompletter Sourcecode in diesem Mono-Repository gespeichert sein, und Sie haben alle selbst erstellten Packages bereits lokal verfügbar.

2.3.3 Module erstellen, verwenden und versionieren

Wie in Abschnitt 2.3 beschrieben, wurde Go mit der Einführung von Go Modules um einen Dependency-Managament-Mechanismus erweitert, den Sie für neue Projekte nutzen sollten.

Zusätzlich zu den Import-Statements im Quellcode, die ein referenziertes Package angeben, wird in Go-Modules-Projekten in einer Meta-Datei die Versionsnummer der Abhängigkeit festgehalten. Diese *go.mod*-Datei im Projekt-Root-Verzeichnis wurde bereits mehrfach angesprochen und hat in einem einfachen Beispiel folgenden Aufbau:

```
module golang.source-fellows.com/training/samples

go 1.14

require github.com/sirupsen/logrus v1.5.0
```

Listing 2.94 Beispiel einer go.mod-Datei im Projekt-Root-Verzeichnis

In der ersten Zeile wird der *Module Path* angegeben, der dem Import Path des Root-Packages entspricht und dem Modul einen eindeutigen Namen gibt. Dieser Name sollte ebenso wie ein Package-Name vollqualifiziert sein, also den Domänennamen enthalten. Es gelten dieselben Regeln, die weiter oben schon für Package-Namen beschrieben wurden.

Neben der Angabe, mit welcher minimalen Go-Version das Modul übersetzt werden kann, enthält die Datei sämtliche benötigen Abhängigkeiten mit Angabe der entsprechenden Version.

Sie können ein Go-Module-basiertes Projekt in einem beliebigen Verzeichnis mit dem Kommando `go mod init` starten. Als Parameter müssen Sie den Module Path Ihres neuen Moduls angeben.

```
go mod init golang.source-fellows.com/training/samples
```

Listing 2.95 Erstellen eines neuen Go-Moduls

Daraufhin werden die bekannte *go.mod*- und eine *go.sum*-Datei in Ihrem aktuellen Projektverzeichnis erstellt. Die *go.sum*-Datei enthält kryptografische Prüfsummen für die verwendeten Abhängigkeiten, mit denen eine Integritätsprüfung durchgeführt werden kann (siehe hierzu auch den Abschnitt zu GOSUMDB, GONOSUMDB in Abschnitt 2.1.7, »Umgebungseinstellungen«).

Abhängigkeiten, die Sie mit `go get` hinzufügen, werden heruntergeladen und mit in die *go.mod*-Datei aufgenommen. Vom Programmcode her spielt es keine Rolle, ob es sich bei der Abhängigkeit um ein Go-Modul handelt oder nicht.

Welche Version der Bibliothek wird verwendet?

`go get` sucht in der Versionsverwaltung nach vorhandenen Tags und versucht, aus deren Namen abzuleiten, welche Version verwendet werden soll.

Die Namen der Tags müssen der Form vX.X.X entsprechen und können auch Zusätze wie `pre` enthalten, z. B. v1.0.0-pre. Standardmäßig wird die Version mit der höchsten Versionsnummer verwendet. Sind keine Tags vorhanden, wird der letzte Commit im sogenannten Default Branch des Repositories benutzt. Welcher Branch das dann ist, muss in der Versionsverwaltung konfiguriert werden.

Durch Angabe einer Versionsnummer bei `go get` können Sie diesen Versionsmechanismus übersteuern, beispielsweise durch den folgenden Aufruf:

```
go get github.com/SourceFellows/somelib@v0.0.1
```

Wenn Sie Ihre Go-Module versionieren möchten, müssen Sie ihnen in der Versionsverwaltung aufsteigende Versionsnummern geben. Das Namensschema sollte sich hierbei an dem *Semantic-Version*-Schema (*https://semver.org/*) orientieren und Versionsnummern im Format v*MAJOR.MINOR.PATCH* verwenden. Eine entsprechende Versions-nummer ist z. B. `v1.0.0`.

Clients bekommen immer dann eine neue Version, wenn Sie Ihren Quellcode in der Versionsverwaltung mit einem neuen Tag versehen haben und mit `go get` in Ihrem Projektverzeichnis ein Update angestoßen haben.

Nach der *Import Compatibility Rule* von Go muss jedes neue Package mit gleichbleibendem Import Path auch abwärtskompatibel gehalten werden. Diese Regel bringt uns beim Einsatz der *Semantic Versions* zu einem Problem:

Eine neue major version einer API stellt eine inkompatible Änderung der Schnittstelle dar. In Go heißt das, dass sich beispielsweise die Signaturen von Funktionen oder Methoden verändert haben. Clients können die neue Version der Bibliothek bzw. das Package nicht ohne Anpassungen weiternutzen. Die *Import Compatibility Rule* wäre in diesem Fall verletzt, da sich die »Schnittstelle« des Import Path verändert hat.

Im vorangegangenen Beispiel in Listing 2.92 sehen Sie z. B. die Version 1.0.0 der Bibliothek *Somelib*. Sie besitzt die Funktion `CallLib()`. Über den entsprechenden Import Path konnten wir die Bibliothek nutzen, wie in Listing 2.93 gezeigt. Besitzt die Version 2.0.0 nun eine inkompatible Änderung, wie in Listing 2.96 dargestellt, ist die *Import Compatibility Rule* verletzt, da der Client angepasst werden müsste, obwohl er den gleichen Import Path nutzt.

```
// Package somelib shows how to build a simple golang library.
package somelib

import "fmt"

// CallLibV2 simple writes a string to the console.
func CallLibV2() {
   fmt.Println("Here is somelib - version 2.0.0")
}
```

Listing 2.96 Version 2 der Somelib

Als Konsequenz müssen Sie in Ihren Module-Namen und den Import Path eine Versionsnummer aufnehmen, um die unterschiedlichen Versionen zu kennzeichnen.

Die Bibliothek muss mit dem Zusatz v2 veröffentlicht werden, und der Client muss, wenn er die Version 2 nutzen möchte, den Import Path entsprechend anpassen. Über Import Aliase können Sie hier den Namen eines eingebundenen Packages konstant halten.

```
// Package main shows how to import a simple library from GitHub.
package main

import somelib "github.com/SourceFellows/somelib/v2"

// Main calls a simple library.
func main() {
   somelib.CallLibV2()
}
```

Listing 2.97 Version 2 eines Somelib Clients

Bei der Entwicklung und Veröffentlichung der Bibliothek können Sie verschiedentlich vorgehen. Ich möchte an dieser Stelle nur die vom Go-Team empfohlene Variante vorstellen, bei der die Version 2 in einem separaten Verzeichnis in der Versionsverwaltung gepflegt wird. Dies hat den Vorteil, dass auch Clients, die keine Go-Module verwenden, weiterarbeiten können.

In Version 1 befand sich die *somelib.go*-Datei direkt im Repository Root unter *https://github.com/SourceFellows/somelib*. Für Version 2 wird die neue *somelib.go*-Datei im Verzeichnis v2 im Repository angelegt und entsprechend entwickelt. Falls eine *go.mod*-Datei vorhanden ist, müssen Sie hier auch den Modul-Namen anpassen.

Version 1

```
module github.com/SourceFellows/somelib
```

zu Version 2

```
module github.com/SourceFellows/somelib/v2
```

Durch das Verschieben der Version 2 in das Verzeichnis *v2* können Clients, unabhängig davon, ob sie Go Modules nutzen oder nicht, den neuen Import Path nutzen (siehe Listing 2.97).

Das Beispiel finden Sie auch in Github unter der angegebenen URL.

2.3.4 Go-Projekte in Go-Module umwandeln

Bis zur Einführung von Go Modules standen verschiedene Möglichkeiten zur Verfügung, um Abhängigkeiten zu verwalten. Es entstanden einige Tools, die diese Aufgabe übernahmen. Jedes dieser Tools hatte entweder einen speziellen Aufbau oder eine Konfigurationsdatei zur Folge.

Wenn Sie ein bestehendes, nicht-Go-Module-basiertes Projekt zu einem Go-Modules-Projekt konvertieren möchten, bietet sich das Tool `go mod` an. Es kann verschiedene Formate von Konfigurationsdateien anderer Abhängigkeitsverwaltungen zu *go.mod*- und *go.sum*-Dateien konvertieren.

Die bekannten Formate bzw. Konfigurationsdateien sind die folgenden:

- GLOCKFILE
- Godeps/Godeps.json

- Gopkg.lock
- dependencies.tsv
- glide.lock
- vendor.conf
- vendor.yml
- vendor/manifest
- vendor/vendor.json

Bei den bekannten Formaten ist eine Konvertierung einfach: Führen Sie hierzu die Initialisierung eines Go-Module-basierten Projekts aus. Geben Sie als Parameter einen passenden Module Path an:

```
go mod init source-fellows.com/training/modsample
```

Das `go mod`-Tool erstellt nun aus den Informationen einer eventuell gefundenen Konfigurationsdatei eine entsprechende *go.mod*-Datei. Im Unterschied zum Ablauf ohne Abhängigkeitsverwaltung werden statt der neuesten Version einer Abhängigkeit die entsprechenden Versionsangaben aus den Konfigurationsdateien verwendet.

Nach einer Konvertierung empfiehlt es sich, `go mod tidy` zum Aufräumen der Abhängigkeiten zu starten. Damit werden transitive Abhängigkeiten aufgelöst und eventuell nicht mehr verwendete bzw. fehlende Abhängigkeiten angepasst.

Sie sollten auf jeden Fall über das Ausführen der Unit-Tests sicherstellen, dass Ihr Modul noch fehlerfrei ausgeführt werden kann und dass sämtliche Abhängigkeiten auch gefunden wurden. Durch die unterschiedlichen Ansätze der Tools kann es vorkommen, dass sich die aufgelösten Abhängigkeiten etwas unterscheiden.

Falls Sie ein Projekt ohne Abhängigkeitsverwaltung bzw. ohne Konfigurationsdatei zu einem Go-Module-basierten Projekt konvertieren, unterscheidet sich der dafür notwendige Ablauf nicht von dem hier dargestellten. Ebenso können Sie über `go mod init` ein neues Projekt starten. Im Gegensatz zu der oben beschriebenen Variante werden dann die Projekt-Quelldateien nach Import-Statements durchsucht und die entsprechenden Abhängigkeiten bestimmt. Da in diesem Fall über die Import-Statements alleine keine Versionsangaben vorliegen, werden die jeweils neusten Versionen aus den entsprechenden Repositories verwendet.

3.5 Die Nebenläufigkeit mit Go – Concurrency

Die Entwicklung von nebenläufigem Code ist in vielen Programmiersprachen schwierig und fehleranfällig. Oftmals wird ein gemeinsamer Speicherbereich genutzt, dessen Zugriff von unterschiedlichen konkurrierenden Clients umständlich und aufwendig synchronisiert werden muss.

Als Entwickler müssen Sie sich mit *Semaphoren*, *Sperren* und *Barrieren* beschäftigen, um mehrere Threads und deren Zugriffe auf diese Speicherbereiche unter Kontrolle zu halten.

In Go ist Nebenläufigkeit ein zentraler Sprachbestandteil, und die benötigten Bausteine sind bereits in der Sprache bzw. der Go-Standardbibliothek integriert. Für die Entwicklung können Sie auf eine einfache, eingebaute Syntax zurückgreifen und müssen keine zusätzlichen Bibliotheken einbinden.

Drei Punkte zeichnen die Nebenläufigkeits-Funktionen in Go besonders aus:

- Grundkonzepte sind einfach zu verstehen.
- Es ist einfach zu benutzen.
- Entstandener Code kann gut erfasst und verstanden werden.

Mit den Konzepten der Nebenläufigkeit können Sie Anwendungen erstellen, die aus unabhängig voneinander laufenden Prozessen bestehen und diese untereinander kommunizieren lassen. Nebenläufigkeit bedeutet nicht zwangsweise Parallelität.

Wenn Ihr Computer nur über eine CPU verfügt, kann Ihre Anwendung trotzdem nebenläufige Prozesse enthalten. In diesem Fall werden sie nicht parallel ausgeführt, sondern zur Ausführung eingeteilt.

Die Konzepte der Nebenläufigkeit können zu einem Softwareentwicklungsmodell führen, das hilft, Sourcecode besser zu strukturieren und Probleme aus der »echten Welt« besser abbilden zu können. Im wahren Leben laufen auch sehr viele Dinge unabhängig voneinander ab und müssen gelegentlich miteinander kommunizieren.

Möchten Sie in Go Nebenläufigkeit entwickeln, sind zwei Punkte wichtig:

- Go-Routinen
- Kommunikationskanäle, die sogenannten *Channels*

3.5.1 Was sind Go-Routines?

Nebenläufige Prozesse werden in der Go-Laufzeitumgebung als sogenannte *Go-Routines* ausgeführt. Im Gegensatz zu anderen Programmiersprachen bzw. Laufzeitumgebungen werden Go-Routinen jedoch nicht als native Betriebssystemprozesse ausgeführt, sondern komplett von der Laufzeitumgebung verwaltet. Nur im Bedarfsfall werden neue Threads im Betriebssystem gestartet.

Vorstellen können Sie sich *Go-Routinen* wie Threads, allerdings sind sie deutlich ressourcenschonender.

In modernen *Java-VMs* wird z. B. pro Thread ein Speicherbereich mit fixer Größe für den Stack reserviert. Somit ist die Anzahl der möglichen Threads durch den Hauptspeicher begrenzt. Die Größe des Speicherbereiches können Sie zwar beim Start der JVM mitgeben, sie ist allerdings dann für jeden Thread gleich groß, egal was er macht. Die Ausgabe in Listing 3.153 zeigt eine typische Ausgabe einer JVM, bei der zu viele Threads gestartet wurden. Wie Sie sehen, konnten auf dieser Hardware mit der genutzten Java-VM-Version insgesamt 10.074 Threads gestartet werden.

```
create native thread 10074
java.lang.OutOfMemoryError: unable to create new native thread
        at java.lang.Thread.start0(Native Method)
        at java.lang.Thread.start(Thread.java:717)
        at Main$1.run(Main.java:28)
        at java.lang.Thread.run(Thread.java:748)
```

Listing 3.153 Ausgabe eines Java-Beispiels beim Anlegen von vielen Threads

Go-Routinen besitzen hingegen eine dynamische Stack-Größe, die zur Laufzeit pro Routine angepasst wird. In Produktivumgebungen sind hunderttausende nebenläufig ausgeführter Go-Routinen keine Seltenheit.

Die hohe Anzahl an Go-Routinen in betriebenen Anwendungen entsteht sicher auch daraus, dass es recht einfach ist, neue Routinen zu starten und Prozesse nebenläufig zu konzipieren.

Starten können Sie eine neue Go-Routine, indem Sie das Schlüsselwort `go` vor einen beliebigen Funktionsaufruf setzen. Im Grunde funktioniert es anlog zu einem `&`-Zeichen in einer *Unix Shell*, mit dem Prozesse im Hintergrund ausgeführt werden können. Nach dem Aufruf mit einem vorangestellten `go` geht es in Ihrem Programm weiter. Auf ein Ergebnis wird nicht gewartet.

```
# ./longRunning.sh &
```

Listing 3.154 Starten eines Kommandos im Hintergrund unter Unix

Nehmen wir das Beispiel aus Listing 3.155 als Ausgangsbasis. Die `main`-Funktion ruft eine weitere Funktion auf, in der innerhalb einer Schleife Text ausgegeben wird. Die Ausgabe erfolgt zeitverzögert mit einer zufälligen Wartezeit.

In diesem Beispiel wird noch keine Go-Routine aktiv gestartet.

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func printOut() {
    for i := 0; i < 10; i++ {
        fmt.Printf("Print %v\n", i)
        time.Sleep(time.Duration(rand.Intn(500)) * time.Millisecond)
    }
}

func main() {
    printOut()
}
```

Listing 3.155 Nicht nebenläufiges Beispiel zur Ausgabe von Text

Wenn Sie nun die `main`-Methode anpassen und vor den Funktionsaufruf das Schlüsselwort `go` schreiben, wird die Ausführung der `printOut`-Funktion in einer separaten Go-Routine gestartet.

```
func main() {
    go printOut()
}
```

Listing 3.156 Starten einer Go-Routine

Bei einem erneuten Start der Anwendung wird Ihnen auffallen, dass keine Ausgabe mehr erscheint. Das hat den einfachen und wichtigen Grund, dass das Beenden der `main`-Funktion die Anwendung mit allen Go-Routinen beendet.

Durch das Schlüsselwort `go` starten Sie im Beispiel eine neue Go-Routine, und Ihre Anwendung läuft weiter. Da es sich allerdings bei diesem Aufruf um das letzte Statement in der `main`-Funktion handelt, wird die Anwendung danach sofort beendet.

Damit Sie doch noch eine Ausgabe erhalten, kann in der `main`-Funktion zusätzlich noch ein `time.Sleep`, wie in der oben gezeigten Schleife, eingebaut werden.

```
func main() {
    go printOut()
    time.Sleep(5 * time.Second)
}
```

Listing 3.157 Starten einer Go-Routine und Warten auf deren Abarbeitung

In diesem einfachen Beispiel sehen wir bereits, dass das Starten von Funktionen als nebenläufiger Prozess einfach ist, aber zusätzlich eine Kommunikation zwischen den Beteiligten benötigt wird. In der `main`-Funktion wäre es deutlich besser, so lange zu warten, bis die Abarbeitung der `printOut`-Funktion beendet ist.

Innerhalb des `sync`-Packages sind einige Synchronisationsfunktionen umgesetzt, mit denen Sie Prozesse synchronisieren könnten. Eine bessere Alternative stellen allerdings die Channels dar, die wir uns im nächsten Abschnitt anschauen.

3.5.2 Wie nutze ich Channels?

Um zwischen einzelnen nebenläufigen Prozessen zu kommunizieren oder diese zu synchronisieren, können Sie in Go sogenannten Channels nutzen. *Channels* sind typisierte Kommunikationskanäle, in die Sie auf der einen Seite Daten hineinschreiben und aus denen Sie auf der anderen Seite Daten wieder auslesen können.

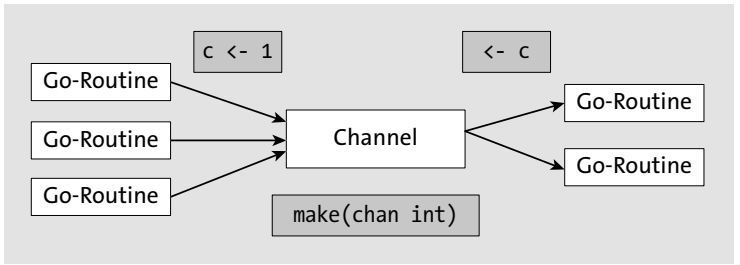

Abbildung 3.28 Go Channels

Die Schreib- und Leseoperationen in bzw. aus einem Channel werden über Pfeile in die jeweilige Richtung der Operation ausgedrückt.

Möchten Sie zum Beispiel den Wert `1` in einen Kanal `c` schreiben, verwenden Sie den Ausdruck `c <- 1`. Das Lesen aus dem Kanal `c` erfolgt mittels `<- c`. Je nachdem, was Sie mit dem gelesenen Wert machen möchten, können Sie ihn entsprechend zuweisen.

Anlegen können Sie einen neuen Kanal, wie in Listing 3.158 gezeigt, mit `make` und der Angabe des entsprechenden Datentyps.

```
//neuen Channel erzeugen
c := make(chan string)
// in den Channel schreiben
c <- "Hello World!"
// aus dem Channel lesen
fmt.Println(<-c)
wert := <-c
// Channel wieder schließen
close(c)
```

Listing 3.158 Grundlegende Arbeit mit Channels

Die Schreib- und Leseoperationen auf einen Kanal sind blockierend. Das heißt, dass eine schreibende Go-Routine so lange blockiert, bis eine zweite Go-Routine den Wert ausliest.

Dieses Verhalten ermöglicht die Synchronisation von Go-Routinen und stellt auch ein Grundprinzip der nebenläufigen Programmierung in Go dar.

Die Konzepte hinter der nebenläufigen Programmierung

Das Konzept der nebenläufigen Programmierung mit der Kommunikation und Synchronisation über Kanäle in Go unterscheidet sich grundlegend von vielen anderen Ansätzen in weiteren Programmiersprachen. Die Ideen dahinter sind allerdings nicht neu, sondern gehen auf Prinzipien z. B. der *communicating sequential processes* (CSP) von Tony Hoare oder auf Edsger Dijkstras *guarded commands* zurück.

Im Internet finden Sie z. B. unter *http://www.usingcsp.com/cspbook.pdf* die Abhandlung von Tony Hoare zu den communicating sequential processes.

Nebenläufige Prozesse sollen nicht auf einen gemeinsamen Speicherbereich zugreifen, der aufwendig synchronisiert und gegen parallelen Zugriff geschützt werden muss, sondern sollen Daten über einen Kommunikationskanal austauschen. Durch die Weitergabe der Daten in einem Channel mit einem Sender und genau einem Empfänger für diese Daten muss darauf keine Synchronisation erfolgen.

Don't communicate by sharing memory, share memory by communicating.
– Rob Pike

Nicht blockierende Zugriffe auf Channels

Mit sogenannten *Buffered Channels* können Sie nicht blockierende Zugriffe auf Channels ermöglichen. Bei der Anlage eines neuen Kanals geben Sie hierzu eine Kapazität mit, wie viele Datensätze zwischengespeichert werden sollen. Erst wenn diese Kapazität aufgebraucht und der Puffer voll ist bzw. der Kanal leer, blockieren die Aufrufe wieder.

Mit dem Einsatz von Buffered Channels verlieren Sie die Synchronisation zwischen lesender und schreibender Go-Routine.

```
c := make(chan int, 5)
```

Listing 3.159 Buffered Channel anlegen

Wir können nun das Beispiel aus Listing 3.157 anpassen und einen Channel zur Synchronisation und zum Datenaustausch nutzen.

Die `printOut`-Methode gibt nun keine Werte mehr direkt über `fmt.Println` aus, sondern schreibt die erzeugten Texte in einen Channel, der in der `main`-Funktion entsprechend initialisiert wird.

Nachdem die `printOut`-Funktion als separate Go-Routine gestartet wurde, können Sie in einer weiteren Schleife die Textwerte aus dem Channel auslesen und ausgeben.

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

var c chan string

func printOut() {
    for i := 1; ; i++ {
        c <- fmt.Sprintf("Print %v", i)
        time.Sleep(time.Duration(rand.Intn(500)) * time.Millisecond)
    }
}

func main() {
    c = make(chan string)
    go printOut()
    for i := 0; i < 10; i++ {
        fmt.Println(<-c)
    }
}
```

Listing 3.160 Kommunikation über Channel zur Synchronisation und zum Datenaustausch

In dem Beispiel verwenden wir im Sender `printOut` eine Endlosschleife zum Schreiben der Werte. Wenn Sie nur eine bestimmte Anzahl an Werten in einen Channel

schreiben möchten, können Sie diesen mit `close` schließen und somit dem Empfänger mitteilen, dass keine weiteren Werte mehr zu erwarten sind.

Auf Empfängerseite können Sie beim Auslesen eines Channels mit einem zweiten Rückgabewert ermitteln, ob der Channel bereits geschlossen wurde.

Das Beispiel können Sie dementsprechend so umstellen:

```
var c chan string

func printOut() {
    for i := 1; i < 10; i++ {
        c <- fmt.Sprintf("Print %v", i)
        time.Sleep(time.Duration(rand.Intn(500)) * time.Millisecond)
    }
    close(c)
}

func main() {
    c = make(chan string)
    go printOut()

    for {
        t, open := <-c
        if open {
            fmt.Println(t)
        } else {
            break
        }
    }
}
```

Listing 3.161 Schließen eines Channels und Prüfung, ob Channel geschlossen ist

Eleganter gehen die Prüfung und das Auslesen eines Channels mit Hilfe des Range-Statements der `for`-Schleife. Sie können den Ausdruck dazu nutzen, die Werte eines Kanals auszulesen. Wird der Channel geschlossen, wird auch die Schleifenbearbeitung beendet.

```
func main() {
    c = make(chan string)
    go printOut()
    for t := range c {
        fmt.Println(t)
    }
}
```

Listing 3.162 Auslesen eines Channels mittels Range-Statement

3.5.3 Typische Vorgehensweisen von Routinen und Channels mit Go

Mit den eingebauten Möglichkeiten der nebenläufigen Programmierung lassen sich gängige Probleme elegant lösen, und es haben sich einige Lösungen mit Go-Routinen und Channels verbreitet.

Ein paar von ihnen wollen wir uns hier anschauen, um Go-Routinen und Channels besser zu verstehen.

Generator Function

In den vorherigen Beispielen haben wir immer einen global definierten Channel verwendet. Da Channels normale Datentypen wie Strings oder Integer sind, können sie auch als Parameter oder Rückgabewerte verwendet werden.

Eine sogenannte *Generator Function* erzeugt intern Werte und schreibt diese in einen Channel. Diesen Channel liefert die Funktion als Rückgabewert, damit Clients daraus lesen können. Durch eine Typisierung des Rückgabewertes auf einen nur lesenden Kanal kann zusätzlich kein Client in diesen Channel Daten schreiben.

Das vorherige Beispiel kann dementsprechend auch wie in Listing 3.163 implementiert werden. Die Funktion `printOut` startet intern eine neue anonyme Funktion als Go-Routine, die in einen Kanal schreibt und diesen als Rückgabewert zurückliefert. Der Kanal ist nicht mehr global definiert, und ein Client kann diesen Kanal direkt auslesen und mit den Werten arbeiten.

```
func printOut() <-chan string {
    c := make(chan string)
    go func() {
        for i := 1; ; i++ {
            c <- fmt.Sprintf("Print %v", i)
            time.Sleep(time.Duration(rand.Intn(500)) * time.Millisecond)
        }
    }()
    return c
}

func main() {
    c := printOut()
    for i := 1; i < 10; i++ {
        fmt.Println(<-c)
    }
}
```

Listing 3.163 Implementierung einer Generator Function

Multiplexer – Fan-in

Wenn Sie von mehreren Service-Aufrufen jeweils einen Channel als Rückgabewert erhalten und diese zusammenführen möchten, können Sie einen sogenannten *Fan-in Multiplexer* einsetzen.

Im Beispiel in Listing 3.164 wird z. B. die oben beschriebene Generator Function zweimal aufgerufen. In einer Schleife sollen die Werte beider Channels ausgegeben werden. Allerdings blockiert die Verarbeitung beider Kanäle, wenn in einem der beiden Kanäle keine Daten vorhanden sind, da es sich jeweils um blockierende Leseoperationen handelt.

```
func main() {
    c1 := printOut()
    c2 := printOut()
    for i := 1; i < 10; i++ {
        fmt.Println(<-c1)
        fmt.Println(<-c2)
    }
}
```

Listing 3.164 Mehrfacher Aufruf der Generator Function

Eine Lösung dieses Problems kann ein Fan-in-Multiplexer sein, den Sie wie in Listing 3.165 erstellen können. Dabei werden in einer `join`-Funktion zwei Go-Routinen gestartet, deren Aufgabe darin besteht, Werte aus einem Eingangskanal in einen Ausgangskanal zu schreiben. Die `join`-Funktion ist in diesem Fall wieder als Generator Function ausgelegt.

Die beiden Ausführungen der `printOut`-Funktionen werden entkoppelt und können vollständig nebenläufig ausgeführt werden.

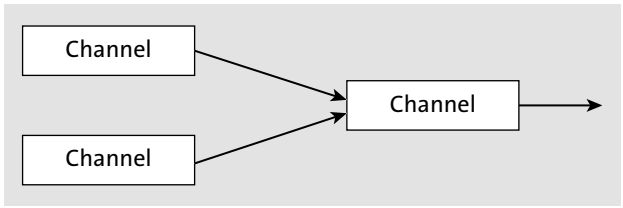

Abbildung 3.29 Go Channel Fan-in-Multiplexer

```
func join(in1, in2 <-chan string) <-chan string {
    out := make(chan string)
    go func() {
        for {
            out <- <-in1
        }
    }()
    go func() {
        for {
            out <- <-in2
        }
    }()
    return out
}

func main() {
    c1 := printOut()
    c2 := printOut()
    c3 := join(c1, c2)
    for i := 1; i < 10; i++ {
        fmt.Println(<-c3)
    }
}
```

Listing 3.165 Zusammenführen von Kanälen

Alternativ können Sie die `join`-Funktion mit einem Select-Statement ausstatten und dafür eine Go-Routine einsparen.

Mit einem Select-Statement kann Ihre Go-Routine auf die Antwort mehrerer Kanäle warten bzw. reagieren. Das Select-Statement ohne `default`-Angabe blockiert so lange, bis Daten aus einem Kanal gelesen werden können. Sind mehrere Kanäle bereit, wird einer zufällig ausgewählt.

```
func join(in1, in2 <-chan string) <-chan string {
    out := make(chan string)
    go func() {
        for {
            select {
            case t := <-in1:
                out <- t
            case t := <-in2:
                out <- t
            }
        }
    }()
    return out
}
```

Listing 3.166 Select-Ausdruck zum Channel auslesen

Timeouts

Durch den Einsatz von Select-Statements und Kanalrückgaben können Sie elegante Implementierungen erstellen, die nach einer gewissen Zeit abgebrochen werden.

Die Funktion `time.After` aus dem `time`-Package liefert z. B. einen Channel, der nach der angegebenen Zeit einen Wert zurückgibt. Bis dahin stehen keine Daten zur Verfügung, und das Lesen aus dem Kanal blockiert.

Wenn Sie nun in einer Schleife zum einen Ihren Datenkanal und zum anderen den Timeout-Kanal auslesen, wird, bis der Timeout-Kanal einen Wert liefert, der andere Kanal ausgewertet. Nach dem angegebenen Timeout-Wert wird abgebrochen.

```
timeout := time.After(3 * time.Second)
for {
    select {
    case t := <-c3:
        fmt.Println(t)
    case <-timeout:
        return
    }
}
```

Listing 3.167 Implementierung eines Timeouts mit Select-Ausdruck

Blockierende Antwort zum Synchronisieren

In manchen Fällen wollen Sie eventuell in Ihrem Sender auf eine Bestätigung des Empfängers warten, und erst dann soll mit der weiteren Verarbeitung fortgefahren werden. Es handelt sich um nichts anderes als eine Synchronisation.

Da es sich bei Channels um normale Typen handelt, können Sie auch innerhalb eines Channels einen Channel zum Empfänger transportieren und ihn Nachrichten schicken lassen.

Im Beispiel in Listing 3.168 sendet die Funktion `printOut` eine Antwort in Form einer `Message` in den Rückgabekanal. Diese Nachricht besteht zum einen wie bisher aus dem eigentlichen Text und zusätzlich aus einem neuen Kanal, der innerhalb der Funktion erstellt wird.

In der Verarbeitung der `printOut`-Methode wird das eigene Ergebnis als `Message` an den Empfänger versendet und anschließend auf irgendeine Nachricht auf dem Kanal gewartet. Der Sender blockiert so lange, bis eine Nachricht eintrifft.

Der Empfänger liest auf der anderen Seite des Kanals die Nachrichten aus und schickt wiederum nach seiner Verarbeitung über den Kanal in der `Message` eine Nachricht an den Sender. Dieser kann nun mit der Verarbeitung fortfahren.

Die beiden Go-Routinen wurden synchronisiert.


```
type Message struct {
    text string
    wait chan interface{}
}

func printOut() <-chan Message {
    finish := make(chan interface{})
    c := make(chan Message)
    go func() {
        for i := 1; ; i++ {
            c <- Message{fmt.Sprintf("Print %v", i), finish}
            time.Sleep(time.Duration(rand.Intn(9000)) * time.Millisecond)
            <-finish
        }
    }()
    return c
}

func main() {
    c1 := printOut()
    c2 := printOut()
    c3 := join(c1, c2)
    for i := 1; i < 20; i++ {
        message1 := <-c3
        message2 := <-c3
        fmt.Println(message1.text)
        fmt.Println(message2.text)
        message1.wait <- ""
        message2.wait <- ""
    }
}
```

Listing 3.168 Message-Objekt für die Antwort mit Synchronisationskanal