Mit Go Cloud-nativ programmieren

Go eignet sich zum Cloud-nativen Entwickeln von Anwendungen. Die Runtime ist schlank und erlaubt, ressourcenschonend zu skalieren.

Von Alex Krause und Markus Zimmermann

Der Erfolg der Sprache Go mag auf den ersten Blick überraschen, denn im Vergleich zu anderen Sprachen beschränken sich Typsystem und funktionale Features auf ein Minimum, auch die Syntax scheint längere Ausdrücke zu fordern. Doch die Vorteile sind andere: Go ist im Cloud-nativen Universum groß geworden, und hier zeigt sich, warum es so erfolgreich ist. Das Geheimnis liegt nicht nur in der Sprache selbst.

Die Cloud Native Computing Foundation definiert Cloud-native Techniken als solche, die es ermöglichen, „skalierbare Anwendungen in modernen dynamischen Umgebungen zu implementieren und zu betreiben“ (siehe ix.de/zuhw). Und hier brilliert Go: Die leichtgewichtige Runtime mit ihren Go-Routinen erlaubt, ressourcenschonend zu skalieren. Ein schneller Compiler, ein verständliches und eingebautes pragmatisches Build und Dependency Management, standardisierte Formatierung und Linting vereinfachen das Entwickeln. Das triviale Deployment als ein direkt ausführbares Binary in einem dadurch leichtgewichtigen Container reduziert die Anforderungen an den Betrieb, unabhängig von der genutzten Infrastruktur. Darüber hinaus ist der größte Teil der Cloud-nativen Tools bereits in Go geschrieben, was zu einer großen Menge an Werkzeugen, Libraries und Dokumentation führt, die den Einstieg in Cloud-native Umgebungen beschleunigen.

Zusammengefasst haben die Entwickler von Go auf eine gute Benutzerfreundlichkeit Wert gelegt, anstatt möglichst viele Sprachfeatures anderer Sprachen nachzubauen. Das Ergebnis ist ein exzellentes Entwickler-Tooling und ein Ökosystem, das perfekt in die Cloud passt.

Merit Money: virtuelles Geld als Belohnung

Die Vorteile von Go als Ökosystem illustriert die Geschichte von Felix. Felix hat ein Start-up gegründet und möchte eine innovative Merit-Money-Applikation als Webprodukt entwickeln. Bei Merit Money handelt es sich um ein Belohnungssystem, bei dem sich die Teammitglieder gegenseitig mit virtuellem Geld belohnen. Dazu erhalten sie jeden Monat einen gewissen Betrag, den sie an ihre Kollegen als Belohnung für gutes Verhalten und Kooperation verteilen sollen. Statt aus den vereinzelten Beobachtungen eines Vorgesetzten erhält jedes Teammitglied somit Feedback und Dankbarkeit aus dem Team in direktem Bezug auf die individuellen und kürzlich erbrachten Leistungen.

Felix hat sich auf Empfehlung eines Freundes für Go als Programmiersprache seines Start-ups entschieden. Er holt sich noch zwei weitere Freunde, Alex und Markus, als Entwickler dazu, die wenig Erfahrung mit Go haben. Bei seinen früheren Projekten hat Felix schon immer Frameworks genutzt, um schneller eine Webapplikation zu bauen. Zusammen mit seinen anderen Entwicklern evaluiert Felix verschiedene HTTP-Frameworks und entscheidet sich für Gin als leichtgewichtiges Webframework.

Dependency Management ist in Go anders als in anderen Sprachen direkt eingebaut. Dennoch muss es für jedes Go-Projekt initialisiert werden. Dafür braucht es die Konfigurationsdatei go.mod, die alle Dependencies des Projekts organisiert. Mit dem Go-CLI lässt sie sich leicht mit `go mod` generieren und verwalten. Der Befehl

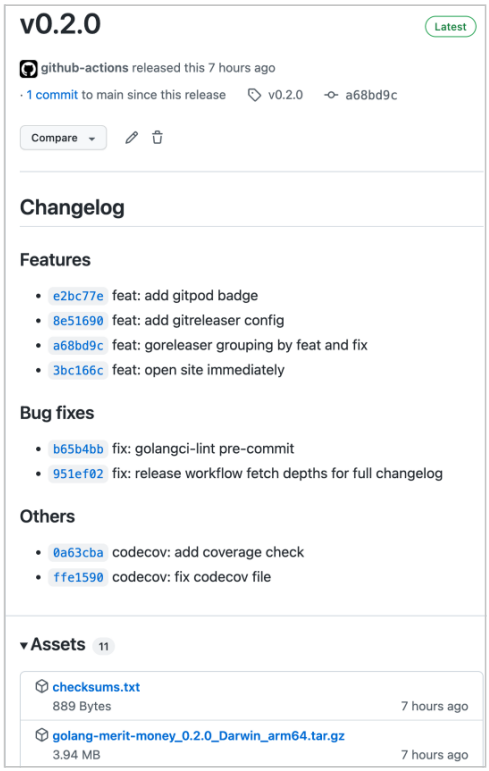
```
go mod init felix-merit-money
```

generiert die go.mod-Datei. Das Go-Projekt ist damit fertig initialisiert. Mit

iX-TRACT

- Go ist Teil eines Ökosystems und Toolings, das an die Cloud angepasst ist.
- Kurze Build- und Startzeiten beschleunigen den Entwicklungszyklus und Cloud-IDEs starten schneller.
- Der einfache Build und fertige Tools für Linting und Release lassen mehr Zeit für das Programmieren.

Die von golangci-lint generierte GitHub-Actions-Annotation zeigt eine ungenutzte Methode an (Abb. 1).



GoReleaser generiert ein GitHub-Release mit Release Notes und Binaries für verschiedene Betriebssysteme (Abb. 2).

```
go get github.com/gin-gonic/gin
```

fügt man Gin als Dependency hinzu. Dabei wird das aktuellste Release als feste Version eingebunden. Auch eine bestimmte Version lässt sich angeben:

```
github.com/gin-gonic/gin@v1.7.7
```

Neue Dependencies sind auch direkt im Code referenzierbar, ohne dass dies in der go.mod erfolgen muss. So genügt

```
import "github.com/gin-gonic/gin"
```

im Quellcode und nachfolgend der Befehl `go mod tidy`, um automatisch die Dependency in die Datei go.mod einzufügen. Der gleiche Befehl entfernt auch nicht mehr benötigte Dependencies und formatiert go.mod.

Meistens hängt eine Dependency von anderen Dependencies ab, den transitiven Abhängigkeiten. Oft bleiben sie unbekannt, da die meisten Package-Manager anderer Programmiersprachen sie nicht explizit in den Konfigurationsdateien auflisten. Das Go-CLI aktualisiert jedoch immer auch alle transitiven Abhängigkeiten in der go.mod-Datei mit der genutzten Version. Dies schafft mehr Klarheit über die Dependencies der Applikation und garantiert deterministische Builds mit weniger Überraschungen, die durch Änderungen in den Dependencies entstehen. Nachdem das Team nun ein Webframework erfolgreich eingebunden hat, kann es jetzt beginnen zu entwickeln.

Continuous Integration ist Pflicht

Felix hat entschieden, dass der Code auf GitHub gehostet werden soll – viele nützliche Integrationen und die hohe Stabilität sprechen dafür. Dank der Leistungsfähigkeit der drei Entwickler steht auch sehr schnell ein kleiner Prototyp von Merit Money und Felix hat eine Vorstellung bei einem potenziellen Kunden geplant. Wie so oft testet er die Applikation vorher auf seinem Rechner und stellt fest, dass sie nicht mehr funktioniert. Die Commit History zeigt an, dass der letzte Commit von Markus war. Auf Nachfrage behauptet Markus, dass die Applikation ohne Probleme auf seinem Rechner funktioniert. Felix hakt weiter nach. Es stellt sich heraus, dass Markus vergessen hat, seinen letzten Fix zu pushen.

Wer ein stabiles Build-System mit einer Continuous-Integration-Pipeline nutzt, kann solche Komplikationen vermeiden. Daher ist die Installation eines dedizierten Build-Servers wie Jenkins, auf dem dann die Projekte gebaut werden, gängige Praxis. GitHub Actions funktioniert ohne das Verwalten dedizierter Server. Die Pipelineaufgaben übernimmt hier GitHub mit eigener Infrastruktur. Es benötigt dafür nur eine Beschreibung der zu verwendenden Actions im jeweiligen GitHub-Projekt. Actions sind die Bausteine für den eigenen Workflow.

Für Go gibt es fertige Actions, die Felix für seinen eigenen Workflow nutzt. Er definiert, dass die Anwendung immer zuerst gebaut und alle Tests ausgeführt werden sollen. Nach jedem Push in das Repository ist somit sichergestellt, dass die Anwendung noch so funktioniert wie

in den Tests spezifiziert. Um Tests zu schreiben oder auszuführen, ist in Go auch weder eine weitere Library noch ein weiteres Tool erforderlich. Durch das testing-Paket lassen sich mit der Go Standard Library Tests schreiben und mit dem Kommando `go test` ausführen.

Als Nächstes möchte Felix wissen, wie gut die Tests sind und ob es noch ungetesteten Code gibt. Die dazu benötigte Testabdeckung will er bei jedem Merge Request messen und anzeigen lassen. Die Coverage lässt sich beim Ausführen von `go test` mit generieren. Um die Coverage zu visualisieren, nutzt er Codecov, das ihm die Informationen für jeden Commit und Push in das Repository anzeigt. Fällt die Testabdeckung unter 75 Prozent, soll die Pipeline fehlschlagen.

Listing 1: Workflow für eine Go-Build- und Testpipeline

```
name: Build and test Go
on: [push]
jobs:
    test:
      runs-on: ubuntu-latest
      steps:
        - name: Set up Go
          uses: actions/setup-go@v3
          with:
            go-version: 1.18.2
        - name: Check out source code
          uses: actions/checkout@v3
        - name: Format Check
          run: test -z "$(gofmt -l .)"
        - name: Test
          run: go test -v -race -coverprofile=coverage.out -covermode=↵
                                                                    atomic ./...
        - name: Upload coverage to Codecov
          uses: codecov/codecov-action@v3
          with:
            files: ./coverage.out
```

Integrierte Codeformatierung erleichtert die Arbeit

Felix macht sich für seine Pipeline eine weitere Eigenschaft von Go zunutze: Codeformatierung ist in das Standard-Tooling integriert. Der Befehl `go fmt` formatiert automatisch alle Codedateien nach den Regeln, die die Go-Entwickler definiert haben. Mit dem Einbau in die Pipeline gehören langwierige Diskussionen über Codeformatierung der Vergangenheit an.

Listing 1 zeigt den fertigen GitHub-Actions-Workflow, der einen Job mit dem Namen „Test“ definiert. Der Job besteht aus mehreren Schritten: Zuerst gilt es, die Go-Runtime einzurichten, die die Action `actions/setup-go` nutzt. Danach ist wie bei jedem Workflowjob der Code mit der `actions/checkout`-Action auszuchecken. Es lassen sich dann alle Operationen auf dem Code ausführen. Der Formatierungscheck nutzt den Befehl `gofmt` und prüft, ob es unformatierte Dateien gibt. Der nächste Schritt führt die Tests aus und generiert die Test-Coverage mit.

Beim Testen wird die Go-Applikation vorher gebaut, sodass sich Build-Fehler unmittelbar erkennen lassen. Das Coverage-Ergebnis lässt sich dann für die `Codecov`-Action nutzen, die die Coverage zu Codecov hochlädt und auch in Merge Requests den Code-Coverage-Status anzeigen kann. Durch diesen simplen Workflow kann Felix nun sicher sein, dass die aktuelle Version auf `main` immer ein Stück lauffähige Software ist.

Felix ist zufrieden mit der kontinuierlichen Integration des Codes. Endlich ist sichergestellt, dass die Tests für jeden Commit durchlaufen. Ihm fällt jedoch immer öfter auf, dass Markus viel Zeit mit Codereviews für die von Alex entwickelten Features verbringt. Auf Nachfrage stellt sich heraus, dass Alex häufig vergisst, sich an einen guten Programmierstil zu halten. Er schreibt gelegentlich zu lange Methoden und nutzt unnötig komplexe Ausdrücke, die den Code schwer verständlich und anpassbar machen. In der Folge braucht Markus länger, sich in Alex' Code einzuarbeiten, und muss ihm im Anschluss Verbesserungsvorschläge unterbreiten. Felix glaubt, dass es möglich sein muss, Markus diese Aufgabe in vielen Fällen abzunehmen, und findet nach kurzer Recherche ein passendes Werkzeug: golangci-lint – einen Linter, der mehrere Go-spezifische Linter miteinander kombiniert und eine einheitliche CLI-Schnittstelle hat.

Felix integriert die passende GitHub Action in die bestehende CI-Pipeline. Kurz darauf schlägt der Build in Alex' Merge Request fehl. In der GitHub-Oberfläche kann er jedoch schnell nachvollziehen, wo die Linter schlechten Code gefunden haben, und die Probleme weitestgehend selbstständig lösen (Abbildung 1). Das spart Markus viel Arbeit. Im Vergleich zu verbreiteten Static Code Analyzers anderer Programmiersprachen ist golangci-lint außerdem sehr schnell, da es selbst in Go geschrieben ist und den Code parallel analysiert.

Nach einer erfolgreichen Demo hat ein potenzieller Kunde Interesse bekundet und möchte Merit Money bei sich testen. Hierzu baut Felix die Binaries für ARM und x86, verpackt sie in Docker Images und lädt sie in einer privaten Docker Registry für den Kunden hoch. Nach einer Woche meldet der Kunde viele Bugs, die Felix noch unbekannt waren. Zusammen mit seinen Entwicklern sind sie schnell gefixt und Felix lädt eine neue Version hoch. Das Spiel wiederholt sich mehrfach und Felix ist es leid, immer ein neues Binary für alle Betriebssysteme zu generieren und dann auf Basis der Änderungen am Code passende Release Notes zu schreiben.

Automatischer Releaseprozess spart Arbeitsschritte

Da macht Alex ihn auf GoReleaser aufmerksam. Das Tool automatisiert den

Listing 2: GitHub-Actions-Workflow zum automatischen Release bei Git-Tag mit GoReleaser

```
on:
    push:
      tags:
        - '*'

name: Upload release assets after tagging
jobs:
    build:
      name: create release
      runs-on: ubuntu-latest
      steps:
        - name: Install Go
          uses: actions/setup-go@v3
          with:
            go-version: 1.18.2
        - name: Checkout code
          uses: actions/checkout@v3
        - name: Run GoReleaser
          uses: goreleaser/goreleaser-action@v3
          with:
            version: latest
            args: release --rm-dist
          env:
            GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

Listing 3: Die Datei .pre-commit-config.yaml definiert, welche Checks gegen den Code ausgeführt werden sollen

```
repos:
- repo: https://github.com/dnephin/pre-commit-golang
    rev: v0.5.0
    hooks:
      - id: go-fmt
      - id: golangci-lint
      - id: go-unit-tests
      - id: go-build
      - id: go-mod-tidy
```

Listing 4: Die Datei .gitpod.yml konfiguriert Gitpod für das Repository

```
image:
    file: .gitpod.Dockerfile
tasks:
    - init: pre-commit install
      command: go run main.go --development
ports:
    - name: Web App
      port: 8080
      onOpen: open-preview
vscode:
    extensions:
      - golang.go
      - casualjim.gotemplate
```

Releaseprozess, indem es Binaries für alle populären Betriebssysteme erstellt, Release Notes generiert und sie zu Releases in GitHub zusammenfasst. Wenn ein Dockerfile existiert, kann GoReleaser auch das Docker Image mit den generierten Binaries bauen und auf eine Docker Registry der Wahl hochladen. Eine passende GitHub Action erleichtert den Einbau in die Pipeline: Hierzu erstellt Felix einen neuen Workflow, der bei jedem neuen Tag läuft und dann mit GoReleaser ein Release erzeugt. Neben dem Workflow braucht es noch eine Konfigurationsdatei, die sich mit dem CLI und dem Befehl `goreleaser init` generieren lässt.

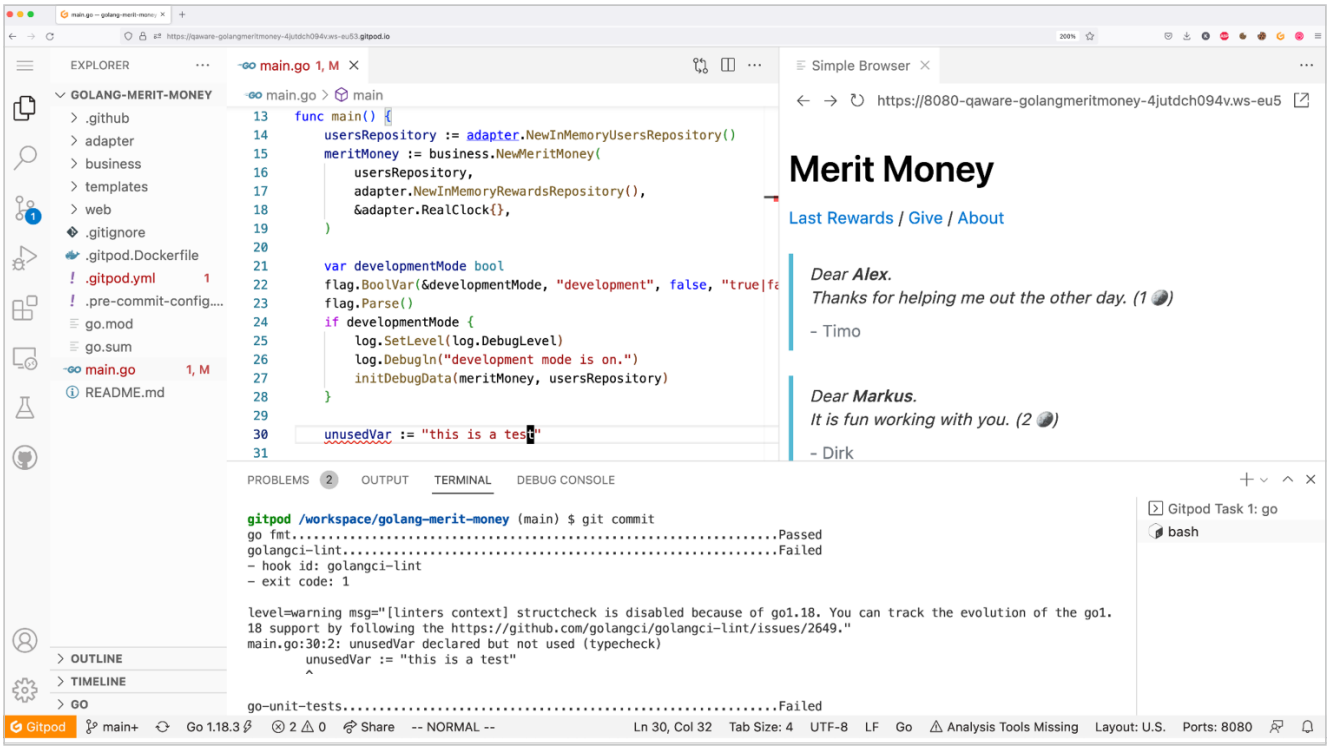
Listing 2 zeigt den GitHub-Actions-Workflow, der bei jedem neuen Tag ein GitHub-Release generiert (Abbildung 2). Wieder muss wie in Listing 1 erst die Go-Runtime und der Code ausgecheckt werden. Danach braucht es nur die vorgefertigte `goreleaser/goreleaser-action@v3`, die dann alle Binaries wie in der Konfiguration beschrieben generiert. Mit dem bei jedem Ausführen des Workflows temporär generierten GitHub-Token erstellt die Action das GitHub-Release. Über unterschiedliche Filter auf der Commit Message sind Commits kategorisierbar und es lassen sich so lesbare Release Notes generieren.

Alex entwickelt ein neues Feature und freut sich über die neue CI-Pipeline, die seinen Code überprüft. Genervt ist er jedoch davon, dass nach jedem `git push` die Pipeline erst nach einiger Verzögerung Fehler meldet, die er dann immer wieder mit Fix Commits beheben muss. Die Commit-Historie für sein Feature ist voll davon. Damit er seinen Code schneller validieren kann, installiert er sich alle Werkzeuge lokal und ruft immer wieder den Linter auf, überprüft die Test-Coverage und lässt den Code automatisiert formatieren. Manchmal vergisst er jedoch einen Aufruf und merkt das erst, wenn die Pipeline fehlschlägt.

Markus beobachtet Alex' Bemühungen für schnelleres Feedback und beschließt, Git Hooks zu verwenden. Git Hooks stoßen das Ausführen von Skripten an, um ein Commit zu validieren. Dies reicht von geänderten Dateien bis zur Commit Message. Mit dem Werkzeug pre-commit lassen sich die Hooks über zentrale Repositorys wiederverwenden. Markus legt hierzu die Datei .pre-commit-config.yaml in das Repository, installiert pre-commit auf Alex' Computer und ruft dann `pre-commit install` auf (Listing 3).

Jedes Mal wenn Alex nun einen neuen Commit mit `git commit` erstellt, laufen die Hooks und brechen den Commit bei mangelnder Qualität ab. Das funktioniert deswegen gut, weil Go schnell kompiliert und die Checks auch in Go geschrieben sind und daher performant laufen. Mit Sprachen wie Java ist Ähnliches kaum zu erreichen: Hier dauert allein das Kompilieren und Starten eines Applikationsservers mit Maven mehrere Sekunden und würde Entwickler immer wieder aus dem Flow reißen. Alex' Quickfix Commits gehören damit nun der Vergangenheit an.

Selina verstärkt das Entwicklerteam. Markus bereitet eine Installationsanleitung für den ersten Arbeitstag vor, um ihr den Einstieg in das Projekt zu erleichtern. Er sammelt hierzu die Kommandos und Schritte, die er verwendet hat, um die einzelnen Werkzeuge wie Compiler, Linter, IDE, IDE-Plug-ins und Tools auf seinem Mac zu installieren. An Selinas erstem Tag stellt Markus jedoch erstaunt fest, dass sie Arch Linux verwendet und seine Anleitung für dieses Betriebssystem nicht funktioniert. Hilfe suchend wendet sich



Die Gitpod-IDE im Browser zeigt rechts die laufende Go-Anwendung und unten das integrierte Terminal (Abb. 3).

Markus an Felix und bittet ihn, die Entwicklerlaptops zu standardisieren.

Works on your Browser: IDE in der Cloud

Selina hat jedoch eine bessere Idee: Die Entwicklungsumgebung soll in der Cloud laufen. Gitpod bietet eine vollwertige IDE im Browser, wobei das Backend der IDE inklusive des Language-Servers, der Plug-ins und des integrierten Terminals in einem Docker-Container in der Cloud läuft (Abbildung 3). Im einfachsten Fall reicht es, das Präfix https://gitpod.io/# vor eine beliebige URL zu einem GitHub-Repository zu stellen und die passende URL aufzurufen. Go ist standardmäßig eingebunden.

Um Gitpod nun optimal für das Projekt zu nutzen, lohnt es sich, eine .gitpod.yml-Datei in das Repository zu legen (Listing 4). Sie definiert das Docker Image, in dem die IDE und das integrierte Terminal laufen sollen, definiert Startkommandos für die Applikation, konfiguriert die IDE mit relevanten Plug-ins für das Projekt und öffnet automatisch den integrierten Browser, um die Applikation anzuzeigen.

Da in Gitpod die Entwicklungsumgebung in der Cloud läuft, spielen das auf dem Rechner der Entwicklerin verwendete Betriebssystem und dessen Hardware praktisch keine Rolle. Solange ein moderner Browser läuft, ist die IDE schnell aufgesetzt.

Weil die Konfiguration mittels Git verwaltet wird, sind auch Änderungen und neue Versionen der Tools schnell im Team verteilt. Git unterstützt zusätzliche benutzerspezifische Standards wie ein bevorzugtes Vim-Plug-in, alternative Tastenkürzel und eigene Dotfiles. Zudem lässt sich Gitpod auch auf einem eigenen Kubernetes installieren.

Go eignet sich durch seine Leichtgewichtigkeit besonders für Cloud-IDEs. Bei jedem Start einer neuen IDE-Session gilt es, alle Dependencies herunterzuladen und die Tools sowie einen frischen Build zu installieren. Go ist in all diesen Punkten schlanker, was zu Startzeiten von wenigen Sekunden führt. Eine Java-Anwendung wie Spring PetClinic braucht hingegen mehr als drei Minuten für den Start in Gitpod.

Fazit

Felix ist schließlich mit seinem Merit Money Service erfolgreich. Wesentlich für diesen Erfolg ist Go. Der einfache Build, das integrierte und einfache Dependency Management, standardisiertes Formatting, die Tools für Linting und Release sowie die Performance bei Pre-Commit Hooks senken die Einstiegshürden. Es bleibt mehr Zeit zum Programmieren.

Bei Java kämpft man im Unterschied dazu nicht selten mit großen Binaries, langsamen Application-Servern, komplizierten Dependency-Management- und Build-Tools sowie schwergewichtigen Static Code Analyzers auf dedizierten Servern. Go macht hier vieles einfacher und beschleunigt gleichzeitig den Start von Cloud-IDEs. (nb@ix.de)

Quellen

Definition der CNCF, Repository mit Code und Beispielen: ix.de/zuhw

ALEX KRAUSE

ist Softwarearchitekt bei QAware. Er liebt es, wenn Cloud-natives Softwareengineering auf moderne Cloud-Plattformen trifft.

MARKUS ZIMMERMANN

ist Senior Software Engineer bei QAware mit Fokus auf Platform Engineering. Nebenbei spricht er als Speaker über Cloud-native Themen und veranstaltet Meetups zu Go.