O'REILLY®

Deutsche Ausgabe

Kubernetes Patterns

Wiederverwendbare Muster zum Erstellen von Cloud-nativen Anwendungen

Bilgin Ibryam
& Roland Huß
Übersetzung von Thomas Demmig

Kubernetes Patterns

Wiederverwendbare Muster zum Erstellen von Cloud-nativen Anwendungen

Bilgin Ibryam und Roland Huß

Deutsche Übersetzung von Thomas Demmig

O'REILLY®

Bilgin Ibryam, Roland Huß

Übersetzung: Thomas Demmig
Lektorat: Sandra Bollenbacher
Korrektorat: Petra Heubach-Erdmann, Düsseldorf
Satz: III-satz, *www.drei-satz.de*
Herstellung: Stefanie Weidner
Umschlaggestaltung: Michael Oréal, *www.oreal.de*
Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

ISBN:
Print 978-3-96009-132-5
PDF 978-3-96010-368-4
ePub 978-3-96010-366-0
mobi 978-3-96010-367-7

1. Auflage 2020

Wieblinger Weg 17
69123 Heidelberg

Dieses Buch erscheint in Kooperation mit O'Reilly Media, Inc. unter dem Imprint »O'REILLY«.
O'REILLY ist ein Markenzeichen und eine eingetragene Marke von O'Reilly Media, Inc. und wird mit Einwilligung des Eigentümers verwendet.

Authorized German translation of the English edition of *Kubernetes Patterns 978-1492050285* © 2019 Bilgin Ibryam and Roland Huß
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Hinweis:
Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.

Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: kommentar@oreilly.de.

5 4 3 2 1 0

Inhalt

Vorwort . . . . . **XI**

Einleitung . . . . . **XIII**

1 Einführung **1**
Der Weg nach Cloud-native 1
Verteilte Primitive 3
Container 5
Pods 6
Services 7
Labels 8
Anmerkungen 10
Namensräume 10
Diskussion 11
Weitere Informationen 12

TEIL I: Grundlegende Patterns **13**

2 Predictable Demands **15**
Problem 15
Lösung 16
Laufzeit-Abhängigkeiten 16
Ressourcen-Profile 18
Pod-Priorität 20
Projekt-Ressourcen 22
Kapazitätsplanung 23
Diskussion 24
Weitere Informationen 24

3 Declarative Deployment 25
Problem 25
Lösung 25
Rollierendes Deployment 27
Fixed Deployment 29
Blue/Green-Release 29
Canary Release 30
Diskussion 31
Weitere Informationen 32

4 Health Probe 33
Problem 33
Lösung 33
Prozess-Health-Checks 34
Liveness-Proben 34
Readiness-Proben 35
Diskussion 36
Weitere Informationen 38

5 Managed Lifecycle 39
Problem 39
Lösung 39
SIGTERM-Signal 40
SIGKILL-Signal 40
Poststart-Hook 41
Prestop-Hook 42
Weitere Lebenszyklus-Elemente 43
Diskussion 44
Weitere Informationen 44

6 Automated Placement 45
Problem 45
Lösung 45
Verfügbare Knoten-Ressourcen 46
Definieren von Container-Ressourcen 47
Platzierungs-Richtlinien 47
Scheduling-Prozess 48
Node Affinity 49
Pod Affinity und Antiaffinity 51
Taints und Tolerations 52
Diskussion 56
Weitere Informationen 57

TEIL II: Verhaltens-Patterns **59**

7 Batch Job **61**
Problem 61
Lösung 62
Diskussion 64
Weitere Informationen 65

8 Periodic Job **67**
Problem 67
Lösung 68
Diskussion 69
Weitere Informationen 70

9 Daemon Service **71**
Problem 71
Lösung 71
Diskussion 74
Weitere Informationen 74

10 Singleton Service **75**
Problem 75
Lösung 76
Out-of-Application Locking 76
In-Application Locking 78
PodDisruptionBudget 80
Diskussion 81
Weitere Informationen 82

11 Stateful Service **83**
Problem 83
Storage 84
Networking 85
Identität 85
Ordinalität 85
Weitere Anforderungen 85
Lösung 86
Storage 87
Networking 88
Identität 90
Ordinalität 90

Weitere Features 91
Diskussion 93
Weitere Informationen 94

12 Service Discovery 95
Problem 95
Lösung 96
Internes Service Discovery 97
Manuelles Service Discovery 100
Service Discovery außerhalb des Clusters 102
Application Layer Service Discovery 106
Diskussion 109
Weitere Informationen 110

13 Self Awareness 111
Problem 111
Lösung 111
Diskussion 115
Weitere Informationen 115

TEIL III: Strukturelle Patterns 117

14 Init Container 119
Problem 119
Lösung 120
Diskussion 124
Weitere Informationen 124

15 Sidecar 125
Problem 125
Lösung 126
Diskussion 128
Weitere Informationen 128

16 Adapter 129
Problem 129
Lösung 129
Diskussion 132
Weitere Informationen 132

17 Ambassador 133
Problem 133

Lösung . . . 133
Diskussion . . . 135
Weitere Informationen . . . 136

TEIL IV: Konfigurations-Patterns . . . 137

18 EnvVar Configuration . . . 139
Problem . . . 139
Lösung . . . 139
Diskussion . . . 142
Weitere Informationen . . . 143

19 Configuration Resource . . . 145
Problem . . . 145
Lösung . . . 145
Diskussion . . . 150
Weitere Informationen . . . 150

20 Immutable Configuration . . . 151
Problem . . . 151
Lösung . . . 151
Docker Volumes . . . 152
Init Container von Kubernetes . . . 153
OpenShift Templates . . . 156
Diskussion . . . 157
Weitere Informationen . . . 158

21 Configuration Template . . . 159
Problem . . . 159
Lösung . . . 159
Diskussion . . . 164
Weitere Informationen . . . 165

TEIL V: Fortgeschrittene Patterns . . . 167

22 Controller . . . 169
Problem . . . 169
Lösung . . . 170
Diskussion . . . 180
Weitere Informationen . . . 181

23 Operator **183**
Problem 183
Lösung 184
Custom Resource Definitions 184
Klassifikation von Controller und Operator 186
Entwickeln und Deployen von Operatoren 189
Beispiel 191
Diskussion 195
Weitere Informationen 196

24 Elastic Scale **197**
Problem 197
Lösung 198
Manuelles horizontales Skalieren 198
Horizontal Pod Autoscaling 199
Vertical Pod Autoscaling 205
Cluster Autoscaling 208
Skalierungs-Stufen 211
Diskussion 213
Weitere Informationen 214

25 Image Builder **215**
Problem 215
Lösung 216
OpenShift Build 217
Knative Build 223
Diskussion 228
Weitere Informationen 229

Nachwort **231**

Index **233**

Vorwort

Als Craig, Joe und ich vor nahezu fünf Jahren mit Kubernetes begannen, war uns allen klar, wie es das Entwickeln und Ausliefern von Software verändern kann. Ich glaube aber nicht, dass wir wussten – oder überhaupt zu glauben wagten –, wie schnell diese Transformation kommen würde. Kubernetes ist nun die Grundlage für das Entwickeln portabler, zuverlässiger Systeme in den großen Public Clouds, in privaten Clouds und auf Bare-Metal-Umgebungen. Aber auch wenn Kubernetes allgegenwärtig geworden ist – mittlerweile können Sie ein Cluster in der Cloud in weniger als fünf Minuten erzeugen und hochfahren –, ist es weit weniger offensichtlich, wie Sie vorgehen sollen, wenn dieses Cluster erst einmal angelegt ist. Es ist toll, dass die Operationalisierung von Kubernetes selbst mit so großen Schritten vorangeht, aber das ist nur ein Teil der Lösung. Es ist die Grundlage, auf der die Anwendungen aufsetzen, und es stellt eine große Bibliothek mit APIs und Tools zum Bauen dieser Anwendungen bereit, aber es bietet dem Anwendungs-Architekten oder Entwickler nur wenig Hilfe oder Anleitung, wie diese diversen Bausteine zu einem vollständigen und zuverlässigen System zusammengesetzt werden können, das die Geschäftsanforderungen und Ziele erfüllt.

Die notwendige Einsicht und Erfahrung, was Sie mit Ihrem Kubernetes-Cluster anstellen können, kann zwar durch frühere Erfahrungen mit ähnlichen Systemen oder durch Versuch und Irrtum erlangt werden, aber das ist sowohl zeitaufwendig als auch der Qualität solcher Systeme nicht zuträglich. Wenn Sie gerade erst lernen, unternehmenskritische Services auf einem System wie Kubernetes auszuliefern, dauert es schlicht zu lange, dies per Versuch und Irrtum zu lernen, und es führt zu sehr realen Problemen durch Downtime und Ausfälle.

Darum ist Bilgins und Rolands Buch so wertvoll. »Kubernetes Patterns« ermöglicht es Ihnen, aus den Erfahrungen zu lernen, die wir in den APIs und Tools codiert haben, die Kubernetes ausmachen. Kubernetes ist eigentlich ein Nebenprodukt der Erfahrung der Community mit dem Bauen und Ausliefern vieler verschiedener, zuverlässiger verteilter Systeme in einer Vielzahl unterschiedlicher Umgebungen. Jedes Objekt und jedes Feature, das zu Kubernetes hinzugefügt wurde, steht für ein grundlegendes Werkzeug, das dazu entworfen und gebaut wurde, eine spezifische Anforderung des Software-Designers zu erfüllen. Dieses Buch erklärt, wie die Kon-

zepte in Kubernetes Probleme aus der Praxis lösen und wie Sie diese Konzepte anpassen und einsetzen, um das System aufzubauen, mit dem Sie heute arbeiten.

Bei der Entwicklung von Kubernetes haben wir immer betont, dass es unser langfristiges Ziel ist, das Entwickeln verteilter Systeme zu einer Übung für eine Einstiegsvorlesung in Informatik zu machen. Wenn es uns gelungen ist, dieses Ziel erfolgreich zu erreichen, sind Bücher wie diese die zugehörigen Lehrbücher. Bilgin und Roland haben die essenziellen Werkzeuge des Kubernetes-Entwicklers erfasst und sie zu Kapiteln eingedampft, die sich leicht erfassen und konsumieren lassen. Wenn Sie dieses Buch durchgearbeitet haben, werden Ihnen nicht nur die Komponenten bekannt sein, die Ihnen in Kubernetes zur Verfügung stehen, sondern auch, warum und wie Sie Systeme mit solchen Komponenten bauen.

– Brendan Burns, Mitbegründer von Kubernetes

Einleitung

In den letzten Jahren hat sich mit dem Aufkommen von Microservices und Containern die Art und Weise, wie wir Software designen, entwickeln und ausführen, deutlich verändert. Heutige Anwendungen sind auf Skalierbarkeit, Elastizität, Fehlertoleranz und schnelle Änderungen optimiert. Getrieben durch neue Prinzipien erfordern diese modernen Architekturen einen anderen Satz an Patterns und Praktiken. Dieses Buch will Entwicklern dabei helfen, Cloud-native Anwendungen mit Kubernetes als Runtime-Plattform zu erschaffen. Schauen wir uns zunächst die beiden wichtigsten Zutaten für dieses Buch an: Kubernetes und Design Patterns.

Kubernetes

Kubernetes ist eine Orchestrierungsplattform für Container. Ihre Ursprünge liegen in den Data Centern von Google, in denen Googles interne Container-Orchestrierungsplattform Borg (*https://ai.google/research/pubs/pub43438*) geschaffen wurde. Google hat Borg viele Jahre genutzt, um seine Anwendungen darauf laufen zu lassen. 2014 entschied sich die Firma dann dazu, ihre Erfahrungen mit Borg in ein neues Open-Source-Projekt namens »Kubernetes« (griechisch für »Steuermann« oder »Pilot«) zu übertragen, und 2015 war dies das erste Projekt, das in die neu gegründete Cloud Native Computing Foundation (CNCF) eingebracht wurde.

Von Anfang an gab es bei Kubernetes eine große Community an Anwendern und die Zahl der Beitragenden wuchs mit unglaublicher Geschwindigkeit. Heute wird Kubernetes als eines der aktivsten Projekte in GitHub betrachtet. Man kann durchaus sagen, dass Kubernetes beim Entstehen dieser Zeilen die am häufigsten genutzte und umfassendste Container-Orchestrierungsplattform ist. Auch bildet es die Grundlage für andere Plattformen, die darauf aufbauen. Das bekannteste dieser Platform-as-a-Service-Systeme ist Red Hat OpenShift, das Kubernetes um diverse zusätzliche Features erweitert, unter anderem um Möglichkeiten, Anwendungen direkt auf der Plattform zu bauen. Das sind nur ein paar der Gründe, warum wir Kubernetes als Referenzplattform für die Cloud-nativen Patterns in diesem Buch gewählt haben.

Dieses Buch geht davon aus, dass Sie ein paar Grundlagen von Kubernetes kennen. In Kapitel 1 wiederholen wir die zentralen Kubernetes-Konzepte und legen das Fundament für die folgenden Patterns.

Design Patterns

Das Konzept der *Design Patterns* (Entwurfsmuster) geht zurück bis in die 1970er Jahre in das Feld der Architektur. Christopher Alexander, Architekt und System-Theoretiker, und sein Team veröffentlichten 1977 das bahnbrechende *A Pattern Language* (Oxford University Press, *https://en.wikipedia.org/wiki/A_Pattern_Language*; deutsch: *Eine Muster-Sprache*, Löcker), das Architekturmuster für Städte, Gebäude und andere Baumaßnahmen enthielt. Einige Zeit später wurde diese Idee von der neu entstehenden Software-Branche übernommen. Das bekannteste Buch in diesem Bereich ist *Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software* von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides – der Gang of Four (Addison-Wesley, *https://de.wikipedia.org/wiki/Entwurfsmuster_(Buch)*). Wenn es um die bekannten Patterns Singleton, Factory oder Delegation geht, liegt das an der Arbeit dieser Autoren. Seitdem wurden viele gute Patterns-Bücher für verschiedene Bereiche und in unterschiedlicher Detailliertheit geschrieben, zum Beispiel *Enterprise Integration Patterns* von Gregor Hohpe und Bobby Woolf (Addison-Wesley, *https://www.enterpriseintegrationpatterns.com*) oder *Patterns für Enterprise Application-Architekturen* von Martin Fowler (mitp Professional, *https://www.martinfowler.com/books/eaa.html*).

Kurz ausgedrückt beschreibt ein Pattern eine *wiederholbare Lösung für ein Problem*.[1] Es ist kein Rezept – denn es gibt keine Schritt-für-Schritt-Anleitung zum Lösen eines Problems, sondern das Pattern beschreibt eine Blaupause für das Lösen einer ganzen Klasse ähnlicher Probleme. So beschreibt zum Beispiel das alexandrinische Pattern »Bierhalle«, wie öffentliche Trinkhallen entworfen werden sollten, sodass »Fremde und Freunde zu Trinkgefährten werden« und sie nicht nur »Zufluchtsstätten für Einsame« sind. Alle Bierhallen, die nach diesem Pattern aufgebaut sind, sehen anders aus, haben aber gemeinsame Charakteristika, wie zum Beispiel offene Nischen für Gruppen von vier bis acht Personen und einen Platz, wo sich einige Hundert Leute mit Bier, Wein und Musik zu Aktivitäten treffen können.

Aber ein Pattern liefert mehr als nur eine Lösung. Es dient auch dazu, eine Sprache zu formen. Die eindeutigen Namen der Patterns führen zu einer dichten, um Nomen zentrierten Sprache, in der jedes Pattern einen eindeutigen *Namen* trägt. Ist

1 Christopher Alexander und sein Team haben die ursprüngliche Bedeutung im Kontext der Architektur wie folgt definiert: »Jedes Muster beschreibt zunächst ein in unserer Umwelt immer wieder auftretendes Problem, beschreibt dann den Kern der Lösung dieses Problems, und zwar so, dass man diese Lösung millionenfach anwenden kann, ohne sich je zu wiederholen.« (*Eine Muster-Sprache*, Christopher Alexander et. al., 1995, Seite x). Wir denken, diese Definition passt auch für die in diesem Buch beschriebenen Patterns, nur dass wir in unseren Lösungen vermutlich keine so große Variabilität haben werden.

die Sprache etabliert, lösen diese Namen automatisch ähnliche mentale Repräsentationen aus, wenn die Leute über diese Patterns sprechen. Reden wir beispielsweise über einen Tisch, geht jeder, der Deutsch spricht, davon aus, dass wir über ein Stück Holz mit vier Beinen und einer Oberfläche reden, auf der Sie Dinge abstellen können. Das Gleiche passiert in der Software-Entwicklung, wenn wir über eine »Factory« sprechen. Im Kontext einer objektorientierten Programmiersprache verbinden wir mit einer »Factory« direkt ein Objekt, das andere Objekte erzeugt. Weil wir sofort die Lösung zum Pattern kennen, können wir uns den noch nicht gelösten Objekten zuwenden.

Es gibt noch andere Charakteristika einer Patterns-Sprache. Patterns sind miteinander verbunden und können sich überlappen, sodass sie zusammen einen Großteil des Problemraums abdecken. Zudem haben Patterns – wie schon im ursprünglichen *Eine Muster-Sprache* dargelegt – nicht immer die gleiche Granularität oder den gleichen Anwendungsbereich. Allgemeinere Patterns decken einen großen Problemraum ab und liefern eine grobe Orientierung, wie ein Problem zu lösen ist. Feingranularere Patterns besitzen einen sehr konkreten Lösungsvorschlag, sind aber nicht so umfassend anwendbar. Dieses Buch enthält alle Arten von Patterns und viele Patterns beziehen sich auf andere oder beinhalten sogar andere Patterns als Teil der Lösung.

Eine weitere Eigenschaft von Patterns ist, dass sie einem strikten Format folgen. Allerdings definiert jeder Autor ein anderes Format und leider gibt es keinen allgemeinen Standard für die Art und Weise, wie Patterns beschrieben werden sollten. Martin Fowler liefert in *Writing Software Patterns (https://www.martinfowler.com/articles/writingPatterns.html)* einen ausgezeichneten Überblick zu den Formaten, die für Patterns-Sprachen verwendet wurden.

Wie dieses Buch aufgebaut ist

Wir haben für dieses Buch ein einfaches Pattern-Format gewählt. Dabei folgen wir keiner bestimmten Sprache zur Beschreibung. Für jedes Pattern nutzen wir die folgende Struktur:

Name
: Jedes Pattern besitzt einen Namen, der gleichzeitig auch die Überschrift des Kapitels ist. Der Name ist der zentrale Aspekt der Patterns-Sprache.

Problem
: Dieser Abschnitt liefert den größeren Kontext und beschreibt den Patterns-Raum im Detail.

Lösung
: In diesem Abschnitt geht es darum, wie das Pattern das Problem auf für Kubernetes spezifische Weise löst. Zudem enthält dieser Abschnitt Verweise auf andere Patterns, die zum aktuellen Pattern entweder in Beziehung stehen oder Teil davon sind.

Diskussion
: Eine Behandlung der Vor- und Nachteile der Lösung für den gegebenen Kontext.

Weitere Informationen
: Dieser letzte Abschnitt enthält zusätzliche Informationsquellen zum Pattern.

Wir haben die Patterns im Buch wie folgt angeordnet:

- Teil I, *Grundlegende Patterns*, behandelt die zentralen Konzepte von Kubernetes. Hier finden sich die zugrunde liegenden Prinzipien und Praktiken für den Aufbau Container-basierter Cloud-nativer Anwendungen.
- Teil II, *Verhaltens-Patterns,* beschreibt Patterns, die auf den grundlegenden Patterns aufbauen und feingranularere Konzepte für das Managen diverser Arten von Container- und Plattform-Interaktionen hinzufügen.
- Teil III, *Strukturelle Patterns,* enthält Patterns zum Organisieren von Containern zu einem *Pod* – der atomaren Einheit der Kubernetes-Plattform.
- Teil IV, *Konfigurations-Patterns,* gibt Einblicke in die diversen Möglichkeiten zur Anwendungskonfiguration in Kubernetes. Dies sind sehr feingranulare Patterns und hier finden sich auch konkrete Rezepte für das Verbinden von Anwendungen mit ihrer Konfiguration.
- Teil V, *Fortgeschrittene Patterns,* ist eine Sammlung komplexerer Konzepte, wie zum Beispiel zum Erweitern der Plattform selbst oder zum Bauen von Container-Images direkt im Cluster.

Ein Pattern muss nicht zwingend genau zu einer Kategorie gehören. Abhängig vom Kontext passt es eventuell in eine Reihe von Kategorien. Jedes Pattern-Kapitel ist in sich abgeschlossen und Sie können Kapitel einzeln und in beliebiger Reihenfolge lesen.

Für wen dieses Buch gedacht ist

Dieses Buch ist für *Entwickler*, die Cloud-native Anwendungen für die Kubernetes-Plattform entwerfen und entwickeln wollen. Es passt am besten zu Lesern, die mit Containern und den Konzepten von Kubernetes prinzipiell vertraut sind und nun den nächsten Schritt gehen wollen. Aber Sie müssen die ganzen Details von Kubernetes nicht kennen, um die Anwendungsfälle und Patterns zu verstehen. Architekten, technische Berater und Entwickler werden alle von den hier beschriebenen, wiederholt einsetzbaren Patterns profitieren.

Dieses Buch baut auf Anwendungsfällen und Lektionen auf, die aus realen Projekten gewonnen wurden. Wir wollen Ihnen dabei helfen, bessere Cloud-native Anwendungen zu bauen – und nicht das Rad neu zu erfinden.

Was Sie lernen werden

In diesem Buch gibt es viel zu entdecken. Manche der Patterns lesen sich auf den ersten Blick vielleicht wie Auszüge aus einer Kubernetes-Anleitung, aber bei einer genaueren Betrachtung werden Sie feststellen, dass die Patterns aus einem konzeptionellen Blickwinkel betrachtet werden, den Sie in anderen Büchern zu diesem Thema nicht finden werden. Andere Patterns werden mit einem anderen Ansatz beschrieben – mit genauen Richtlinien für sehr konkrete Probleme, wie zum Beispiel in *Konfigurations-Patterns* in Teil IV.

Unabhängig von der Granularität der Patterns werden Sie alles kennenlernen, was Kubernetes für jedes einzelne Pattern anbietet – mit vielen Beispielen, um die Konzepte zu illustrieren. All diese Beispiele wurden getestet und wir werden Ihnen in »Die Verwendung von Codebeispielen« auf Seite XVIII erklären, wie Sie an den vollständigen Quellcode gelangen.

Bevor wir in die Thematik richtig eintauchen, schauen wir uns an, was dieses Buch *nicht* ist:

- Dieses Buch ist keine Anleitung zum Aufsetzen eines Kubernetes-Clusters. Jedes Pattern und jedes Beispiel geht davon aus, dass Kubernetes bei Ihnen schon läuft. Es gibt eine Reihe von Möglichkeiten, die Beispiele auszuprobieren. Sind Sie daran interessiert, zu lernen, wie Sie ein Kubernetes-Cluster aufsetzen, empfehlen wir *Managing Kubernetes* von Brendan Burns und Craig Tracey (O'Reilly, *https://oreil.ly/2HoadnU*). Auch das *Kubernetes Cookbook* von Michael Hausenblas und Sébastien Goasguen (O'Reilly, *https://bit.ly/2FTgJzk*) enthält Rezepte für das Aufsetzen eines Kubernetes-Clusters von Grund auf.
- Dieses Buch ist keine Einführung in Kubernetes und auch keine Referenz. Wir stellen viele Kubernetes-Features vor und erklären manche auch im Detail, aber wir fokussieren uns auf die Konzepte hinter diesen Features. In Kapitel 1 liefern wir Ihnen eine kurze Wiederholung der Grundlagen von Kubernetes. Suchen Sie nach einem umfassenderen Buch für den Einsatz von Kubernetes, möchten wir Ihnen *Kubernetes in Action* von Marko Lukša (Manning Publications) sehr ans Herz legen.

Das Buch ist in einem lockeren Stil geschrieben und ähnelt eher einer Reihe von Essays, die unabhängig voneinander gelesen werden können.

Konventionen

Wie schon erwähnt, bilden Patterns eine Art einfache, untereinander verbundene Sprache. Um dieses Netz an Patterns hervorzuheben, ist jedes Pattern *kursiv* geschrieben (zum Beispiel *Sidecar*). Ist ein Pattern wie ein zentrales Konzept von Kubernetes benannt (zum Beispiel *Init Container* oder *Controller*), nutzen wir diese Formatierung nur, wenn wir uns direkt auf das Pattern beziehen. Wo es sinnvoll

ist, verweisen wir auch auf Pattern-Kapitel, damit Sie leichter dorthin wechseln können.

Zudem nutzen wir folgende Konventionen:

- Alles, was Sie in eine Shell oder einen Editor eintippen können, ist in `Nichtproportionalschrift` wiedergegeben.
- Die Namen von Kubernetes-Ressourcen werden immer so genutzt, wie sie in Kubernetes selbst vorkommen. Hat eine Ressource einen kombinierten Namen wie ConfigMap, nutzen wir diesen statt des sprachlich korrekteren »Config Map«, um klarer zu machen, dass wir uns auf ein Kubernetes-Konzept beziehen.
- Manchmal ist der Name einer Kubernetes-Ressource identisch zu einem allgemeineren Konzept wie »Node«. In diesem Fall verwenden wir den Ressourcen-Namen nur, wenn wir uns auf die Ressource selbst beziehen.

Dieses Element kennzeichnet einen Tipp oder Vorschlag.

Dieses Element kennzeichnet einen allgemeinen Hinweis.

Dieses Element kennzeichnet eine Warnung oder einen Achtungshinweis.

Die Verwendung von Codebeispielen

Jedes Pattern ist durch vollständig ausführbare Beispiele ergänzt, die Sie auf der zugehörigen Webseite *(https://k8spatterns.io)* finden. Sie können den Link für jedes Beispiel eines Patterns dem Abschnitt »Weitere Informationen« jedes Kapitels entnehmen.

Der Abschnitt »Weitere Informationen« enthält auch viele Links auf zusätzliche Informationen, die mit dem Pattern in Verbindung stehen. Wir halten diese Liste im Beispiel-Repository aktuell. Änderungen an den Links werden auch auf Twitter *(https://twitter.com/k8spatterns)* gepostet.

Der Quellcode für alle Beispiele aus diesem Buch steht auf GitHub *(https://github.com/k8spatterns)* zur Verfügung. Das Repository und die Website enthalten zudem Anleitungen und Links, wie Sie an ein Kubernetes-Cluster gelangen können, um

die Beispiele auszuprobieren. Gehen Sie die Beispiele durch, schauen Sie sich auch die zugehörigen Ressourcen-Dateien an. Sie enthalten viele nützliche Kommentare, die dabei helfen können, den Beispielcode besser zu verstehen.

Viele Beispiele nutzen einen REST-Service namens *random-generator*, der bei einem Aufruf Zufallszahlen zurückliefert. Er ist nur dazu gedacht, gut mit den Beispielen in diesem Buch zusammenzuarbeiten. Sein Quellcode findet sich ebenfalls auf GitHub *(https://github.com/k8spatterns/random-generator)* und sein Container-Image `k8spatterns/random-generator` ist auf Docker Hub gehostet.

Zum Beschreiben von Ressourcen-Feldern nutzen wir eine JSON-Path-Notation. So verweist beispielsweise `.spec.replicas` auf das Feld `replicas` des Abschnitts `spec` der Ressource.

Finden Sie ein Problem im Beispielcode oder in der Dokumentation oder haben Sie eine Frage, dürfen Sie gerne ein Ticket im GitHub Issue Tracker *(https://github.com/k8spatterns/examples/issues)* öffnen. Wir haben ein Auge auf diese Issues und beantworten dort auch gerne Fragen.

Der gesamte Beispielcode steht unter der Lizenz Creative Commons Attribution 4.0 (CC BY 4.0) *(https://creativecommons.org/licenses/by/4.0)*. Der Code darf frei verwendet werden und Sie können ihn gerne weitergeben und für kommerzielle und nicht kommerzielle Projekte anpassen. Aber Sie sollten auf das Buch verweisen, wenn Sie das Material kopieren oder weiterverteilen.

Dabei kann es sich entweder um eine Referenz auf das Buch mit Titel, Autor, Verlag und ISBN handeln, zum Beispiel »Kubernetes Patterns von Bilgin Ibryam und Roland Huß (O'Reilly). Copyright 2020 Bilgin Ibryam und Roland Huß, 978-3-86490-726-5«. Alternativ nehmen Sie einen Link auf die begleitende Website *(https://k8spatterns.io)*. zusammen mit einem Copyright-Hinweis und einem Link auf die Lizenz auf.

Eigene Code-Beiträge finden wir auch toll! Wenn Sie der Meinung sind, unsere Beispiele verbessern zu können, möchten wir gerne von Ihnen hören. Öffnen Sie einfach ein GitHub-Issue oder erzeugen Sie einen Pull-Request, damit wir miteinander reden können.

Wie Sie uns erreichen können

Mit Anmerkungen, Fragen oder Verbesserungsvorschlägen zu diesem Buch können Sie sich gerne an den Verlag wenden: *kommentar@oreilly.de*

Danksagungen

Das Erstellen dieses Buches war wie eine lange Reise, die über zwei Jahre ging, und wir möchten all unseren Reviewern dafür danken, dass wir auf dem richtigen Weg blieben. Ein besonderer Dank geht an Paolo Antinori und Andrea Tarocchi, die

uns während der ganzen Reise begleitet und geholfen haben. Ein großer Dank auch an Marko Lukša, Brandon Philips, Michael Hüttermann, Brian Gracely, Andrew Block, Jiri Kremser, Tobias Schneck und Rick Wagner, die uns mit ihrem Wissen und ihren Ratschlägen halfen. Und schließlich ein großes Dankeschön an unsere Lektorinnen und Lektoren Virginia Wilson, John Devins, Katherine Tozer, Christina Edwards und all die anderen wunderbaren Leute bei O'Reilly, die uns dabei geholfen haben, das Buch auch ins Ziel zu bringen.

KAPITEL 1

Einführung

In diesem einführenden Kapitel legen wir die Grundlagen für den Rest des Buches, indem wir ein paar der zentralen Kubernetes-Konzepte erklären, die zum Designen und Implementieren von Container-basierten Cloud-nativen Anwendungen genutzt werden. Es ist ausgesprochen wichtig, diese neuen Abstraktionen und die dazugehörigen Prinzipien und Patterns aus diesem Buch zu verstehen, um verteilte Anwendungen zu bauen, die automatisiert von Cloud-nativen Plattformen verteilt werden.

Dieses Kapitel ist keine Voraussetzung für das Verstehen der später beschriebenen Patterns. Leserinnen und Leser, die mit den Konzepten von Kubernetes vertraut sind, können es überspringen und direkt zu den für sie interessanten Patterns-Kategorien springen.

Der Weg nach Cloud-native

Die beliebteste Anwendungs-Architektur auf Cloud-nativen Plattformen wie Kubernetes ist der Microservice-Stil. Diese Technik der Softwareentwicklung geht die Komplexität von Software an, indem sie die einzelnen Aspekte der Geschäftsvorgänge modularisiert und Entwicklungs-Komplexität gegen operative Komplexität eintauscht.

In der Microservice-Bewegung gibt es einen deutlichen Anteil an Theorie und unterstützenden Techniken zum Erstellen neuer Microservices oder für das Aufteilen von Monolithen in Microservices. Die meisten dieser Praktiken basieren auf dem Buch *Domain-Driven Design* von Eric Evans (Addison-Wesley, *https://dddcommunity.org/book/evans_2003*) und den Konzepten der Bounded Contexts und Aggregate. *Bounded Contexts* kümmern sich um große Modelle, indem sie sie in unterschiedliche Komponenten unterteilen, während *Aggregate* dabei helfen, die Bounded Contexts in Module mit definierten Transaktionsgrenzen zu unterteilen. Aber neben diesen Überlegungen aus Business-Sicht gibt es für jedes verteilte System – ob es nun auf Microservices basiert oder nicht – eine Reihe technischer Aspekte rund um seine Organisation, Struktur und das Laufzeitverhalten.

Container und Container-Orchestrierer wie Kubernetes stellen viele neue Primitive und Abstraktionen bereit, um die Probleme mit verteilten Anwendungen anzugehen, und hier beschreiben wir die verschiedenen Möglichkeiten, über die Sie nachdenken sollten, wenn Sie ein verteiltes System auf Kubernetes bringen wollen.

Im Buch gehen wir Container- und Plattform-Interaktionen an, indem wir die Container als Blackboxes behandeln. Aber wir haben diesen Abschnitt geschrieben, um hervorzuheben, wie wichtig es ist, *was* in diesen Containern ist. Container und Cloud-native Plattformen bieten Ihren verteilten Anwendungen unglaubliche Vorteile, aber wenn Sie in Ihre Container nur Müll stecken, werden Sie auch nur Müll skalieren. In Abbildung 1-1 sehen Sie die verschiedenen Fähigkeiten, die erforderlich sind, um gute Cloud-native Anwendungen zu schreiben.

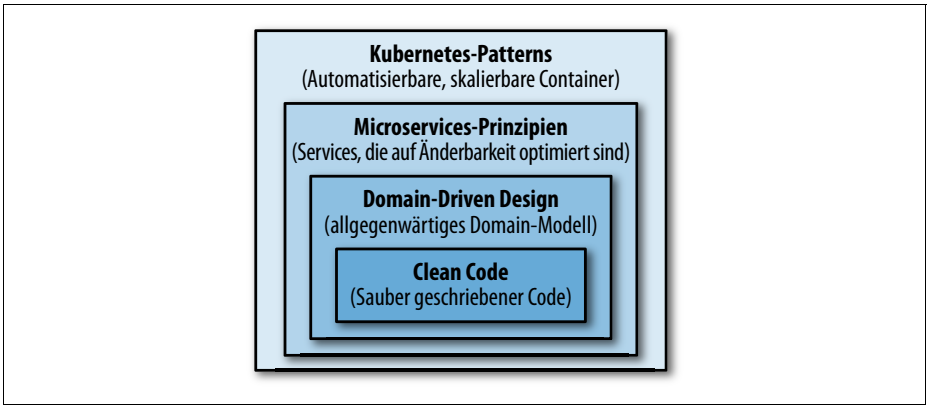

Abbildung 1-1: Der Weg nach Cloud-native

Aus der Vogelperspektive betrachtet gibt es in einer Cloud-nativen Anwendung eine Reihe von Abstraktionsschichten, für die unterschiedliche Design-Überlegungen erforderlich sind:

- Auf unterster *Codeebene* spielt jede von Ihnen definierte Variable, jede Methode und jede instanziierte Klasse in der langfristigen Wartung der Anwendung eine Rolle. Es ist egal, welche Container-Technologie oder welche Orchestrierungs-Plattform Sie nutzen – das Entwicklungs-Team und die von ihm erzeugten Artefakte haben den größten Einfluss. Es ist wichtig, Entwickler aufzubauen, deren Ziel es ist, Clean Code zu schreiben, die richtige Menge an automatisierten Tests einzusetzen, fortlaufend zu refaktorieren, um die Qualität des Codes zu verbessern, und die in ihrem Herzen Code-Handwerker sind.
- Beim *Domain-Driven Design* geht es darum, ein Software-Design aus Business-Perspektive mit der Intention anzugehen, die Architektur so nahe wie möglich an der Realität zu orientieren. Dieser Ansatz funktioniert am besten für objektorientierte Programmiersprachen, aber es gibt auch andere gute Wege, Software für Probleme aus der realen Welt zu modellieren und zu entwerfen. Ein Modell mit den richtigen Business- und Transaktions-Grenzen,

einfach zu nutzenden Schnittstellen und umfangreichen APIs ist die Grundlage für ein späteres erfolgreiches Containerisieren und Automatisieren.

- Der *Microservices-Architekturstil* hat sich sehr schnell zum Standardvorgehen entwickelt und er bietet wertvolle Prinzipien und Praktiken für das Designen von sich ändernden verteilten Applikationen. Das Anwenden dieser Prinzipien ermöglicht es Ihnen, Implementierungen zu schaffen, die auf Skalierbarkeit, Resilienz und Änderbarkeit optimiert sind, was für heutige, moderne Software übliche Anforderungen sind.
- Container wurden sehr schnell als der Standardweg zum Verpacken und Ausführen verteilter Anwendungen akzeptiert. Das Erstellen modularer, wiederverwendbarer Container, die gute Cloud-native Mitbürger sind, ist eine weitere grundlegende Voraussetzung. Mit einer wachsenden Zahl von Containern entsteht in jeder Organisation die Notwendigkeit, sie mit effektiveren Methoden und Werkzeugen zu managen. *Cloud-native* ist ein recht neuer Begriff, mit dem die Prinzipien, Patterns und Tools zum Automatisieren von containerisierten Microservices im großen Maßstab beschrieben werden. Wir nutzen *Cloud-native* parallel zu *Kubernetes* – der beliebtesten aktuell verfügbaren Cloud-nativen Open-Source-Plattform.

In diesem Buch gehen wir nicht weiter auf Clean Code, Domain-Driven Design oder Microservices ein. Wir konzentrieren uns nur auf die Patterns und Praktiken rund um das Orchestrieren von Containern. Aber damit diese Patterns effektiv eingesetzt werden können, muss Ihre Anwendung aus dem Inneren heraus gut entworfen sein, indem Clean-Code-Praktiken, Domain-Driven Design, Microservices-Patterns und andere relevante Design-Techniken angewendet werden.

Verteilte Primitive

Um zu erläutern, was wir mit neuen Abstraktionen und Primitiven meinen, vergleichen wir sie hier mit der wohlbekannten objektorientierten Programmierung (OOP) und insbesondere Java. Im OOP-Universum haben wir Konzepte wie Klasse, Objekt, Paket, Vererbung, Kapselung und Polymorphismus. Dann liefert die Java-Runtime spezifische Features und Garantien zum Managen des Lebenszyklus unserer Objekte und der Anwendung im Ganzen.

Die Java-Sprache und die Java Virtual Machine (JVM) bieten lokale In-Process-Bausteine für das Erstellen von Anwendungen. Kubernetes fügt dieser bekannten Denkweise eine ganz neue Dimension hinzu, indem es einen neuen Satz an verteilten Primitiven und eine weitere Runtime für das Bauen verteilter Systeme anbietet, die über viele Knoten und Prozesse verteilt sind. Mit Kubernetes sind Sie nicht mehr nur auf lokale Primitive angewiesen, um das gesamte Anwendungsverhalten zu implementieren.

Sie müssen immer noch die objektorientierten Bausteine nutzen, um die Komponenten der verteilten Anwendung zu bauen, aber Sie können auch Kubernetes-Primitive für manche der Anwendungsaspekte einsetzen. Tabelle 1-1 zeigt, wie verschiedene Entwicklungs-Konzepte mit lokalen und verteilten Primitiven unterschiedlich umgesetzt werden.

Tabelle 1-1: Lokale und verteilte Primitive

Konzept	Lokales Primitiv	Verteiltes Primitiv
Verhalten kapseln	Klasse	Container-Image
Verhaltens-Instanz	Objekt	Container
Wiederverwendbare Einheit	*.jar*	Container-Image
Komposition	Klasse A enthält Klasse B	Sidecar-Pattern
Vererbung	Klasse A erweitert Klasse B	`FROM` Super-Image eines Containers
Deployment-Einheit	*.jar/.war/.ear*	Pod
Buildtime/Runtime-Isolierung	Modul, Paket, Klasse	Namespace, Pod, Container
Initialisierungs-Vorbedingungen	Konstruktor	Init Container
Postinitialisierungs-Trigger	Init-Methode	`postStart`
Pre-Destroy-Trigger	Destroy-Methode	`preStop`
Cleanup-Prozedur	`finalize()`, Shutdown-Hook	Defer-Container[1]
Asynchrone und parallele Ausführung	`ThreadPoolExecutor`, `ForkJoinPool`	Job
Wiederholte Aufgabe	`Timer`, `ScheduledExecutorService`	CronJob
Hintergrundaufgabe	Daemon-Thread	DaemonSet
Konfigurationsmanagement	`System.getenv()`, `Properties`	ConfigMap, Secret

1 Defer-(oder De-Init-)Container sind noch nicht implementiert, aber es gibt ein Proposal *(http://bit.ly/2TegEM7)*, um dieses Feature in zukünftigen Versionen von Kubernetes mit aufzunehmen. Wir beschreiben Lifecycle-Hooks in Kapitel 5.

Die In-Process-Primitive und die verteilten Primitive haben Gemeinsamkeiten, aber sie sind nicht direkt vergleich- und austauschbar. Sie arbeiten auf unterschiedlichen Abstraktionsebenen und haben verschiedene Vorbedingungen und Garantien. Manche Primitive sind dazu gedacht, gemeinsam eingesetzt zu werden. So müssen Sie beispielsweise immer noch Klassen nutzen, um Objekte zu erstellen und diese in Container-Images zu packen. Aber manche anderen Primitive, wie zum Beispiel CronJobs in Kubernetes, können vollständig durch das `ExecutorService`-Verhalten in Java ersetzt werden.

Schauen wir uns als Nächstes ein paar verteilte Abstraktionen und Primitive von Kubernetes an, die für Anwendungsentwickler besonders von Interesse sind.

Container

Container sind die Bausteine für Kubernetes-basierte Cloud-native Anwendungen. Stellen Sie einen Vergleich mit OOP und Java an, sind Container-Images wie Klassen und Container wie Objekte. Wie Sie Klassen erweitern können, um sie wiederzuverwenden und das Verhalten anzupassen, können Sie Container-Images nutzen, die andere Container-Images erweitern, um sie wiederzuverwenden und das Verhalten anzupassen. Und wie Sie Objektkomposition einsetzen und Funktionalität verwenden können, können Sie Container zusammenfügen, indem Sie sie in einen Pod stecken und miteinander arbeiten lassen.

Gehen wir mit dem Vergleich noch weiter, wäre Kubernetes wie die JVM, nur dass es über viele Hosts verteilt ist und verantwortlich wäre für das Ausführen und Managen der Container.

Init Container wären so etwas wie Objekt-Konstruktoren; DaemonSets entsprechen Daemon-Threads, die im Hintergrund laufen (wie beispielsweise der Java Garbage Collector). Ein Pod entspräche einem Inversion of Control(IoC)-Kontext (zum Beispiel Spring Framework), in dem sich mehrere laufende Objekte einen Managed Lifecycle teilen und direkt aufeinander zugreifen können.

Die Parallelen gehen nicht viel weiter, aber der Punkt ist, dass Container eine zentrale Rolle in Kubernetes spielen, und das Erstellen modularisierter, wiederverwendbarer Container-Images mit jeweils einem Zweck ist für den langfristigen Erfolg jedes Projekts und sogar des Container-Ökosystems im Ganzen ausgesprochen wichtig. Abgesehen von den technischen Eigenschaften eines Container-Image, die Packaging und Isolation bieten – was repräsentiert ein Container und was ist sein Zweck im Kontext einer verteilten Anwendung? Hier ein paar Vorschläge, wie Sie Container betrachten können:

- Ein Container-Image ist die Funktionalitätseinheit, die einen einzigen Zweck angeht.
- Ein Container-Image wird von einem Team verantwortet und besitzt einen eigenen Release-Zyklus.
- Ein Container-Image ist in sich abgeschlossen, definiert seine Runtime-Abhängigkeiten und bringt sie mit.
- Ein Container-Image ist nicht veränderbar, und nachdem es gebaut wurde, wird es nicht mehr angefasst – es wird nur noch konfiguriert.
- Ein Container-Image hat definierte Runtime-Abhängigkeiten und Ressourcen-Anforderungen.
- Ein Container-Image hat wohldefinierte APIs, um seine Funktionalität nach außen anzubieten.
- Ein Container führt im Allgemeinen einen einzelnen Unix-Prozess aus.
- Ein Container ist verwerfbar und lässt sich jederzeit sicher hoch- oder herunterskalieren.

Neben all diesen Eigenschaften ist ein ordentliches Container-Image modular. Es ist parametrisierbar und für eine Wiederverwendung in unterschiedlichen Umgebungen geschaffen. Aber es ist auch für unterschiedliche Anwendungsfälle anpassbar. Kleine, modulare und wiederverwendbare Container-Images führen langfristig zum Entstehen spezialisierterer und stabilerer Container-Images – so wie das bei guten wiederverwendbaren Bibliotheken in der Welt der Programmiersprachen geschieht.

Pods

Schauen Sie sich die Eigenschaften von Containern an, können Sie sehen, dass sie perfekt zum Implementieren der Prinzipien von Microservices geeignet sind. Ein Container-Image stellt eine einzelne Funktionalitätseinheit bereit, gehört zu einem einzelnen Team, besitzt einen eigenen Release-Zyklus und stellt Deployment- und Runtime-Isolation bereit. Meistens entspricht ein Microservice einem Container-Image.

Aber die meisten Cloud-nativen Plattformen bieten ein weiteres Primitiv zum Managen des Lebenszyklus einer Gruppe von Containern an – in Kubernetes als Pod bezeichnet. Ein *Pod* ist die atomare Einheit zum Schedulen, Deployen und für die Runtime-Isolation einer Gruppe von Containern. Alle Container in einem Pod werden immer auf dem gleichen Host geschedult, gemeinsam deployt – sei es aus Gründen des Skalierens oder wegen einer Host-Migration –, und können sich Namensräume für das Dateisystem, Netzwerk und Prozesse teilen. Dieser gemeinsame Lebenszyklus erlaubt es den Containern in einem Pod, untereinander über das Dateisystem oder auf Netzwerkebene über localhost oder Host-Interprozess-Kommunikationsmechanismen zu interagieren, wenn das (zum Beispiel aus Performancegründen) gewünscht ist.

Wie Sie in Abbildung 1-2 sehen, entspricht ein Microservice während der Entwicklung und beim Bauen einem Container-Image, das ein Team entwickelt und releast. Aber zur Laufzeit wird ein Microservice durch einen Pod repräsentiert – die Einheit zum Deployen, Schedulen und Skalieren. Die einzige Möglichkeit, einen Container laufen zu lassen – sei es zum Skalieren oder beim Migrieren –, ist über die Pod-Abstrahierung. Manchmal enthält ein Pod mehr als einen Container. Ein solches Beispiel ist ein containerisierter Microservice, der zur Laufzeit einen Hilfs-Container einsetzt, wie das später in Kapitel 15 gezeigt wird.

Container und Pods und ihre einmaligen Eigenschaften ermöglichen einen neuen Satz an Patterns und Prinzipien für das Designen von auf Microservices basierenden Anwendungen. Wir haben uns manche der Eigenschaften gut entworfener Container angeschaut – jetzt wollen wir das Gleiche für einen Pod tun:

- Ein Pod ist die atomare Einheit beim Schedulen. Der Scheduler versucht also, einen Host zu finden, der die Anforderungen aller Container im Pod erfüllt (es gibt ein paar spezielle Aspekte in Bezug auf Init Container, die wir in Kapitel 14