Kubernetes

Das Praxisbuch für Entwickler und DevOps-Teams

Kapitel 2
Einführung in Kubernetes

»Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications.«
– kubernetes.io

Container zu entwickeln und auf Ihrem Laptop laufen zu lassen, ist mit ein wenig Know-how machbar und einfach. Hunderte von Containern über mehrere Host-Systeme zu betreiben, sie nach Bedarf zu skalieren und dabei keine Downtime zu riskieren, ist dabei schon um einiges komplizierter.

Kubernetes (K8s) wurde entwickelt, um diese Probleme anzugehen und zu lösen. Der Name stammt aus dem Griechischen und bedeutet Steuermann. Daher kommt auch das Logo von Kubernetes. Von Google entwickelt und als Open-Source-Projekt weitergetrieben, ist es heute aus vielen Unternehmen nicht mehr wegzudenken.

Bevor ich Sie dabei begleite, Ihr erstes eigenes Kubernetes-Cluster auf Ihrem Rechner zu installieren, tauchen wir ab in die Grundlagen von Kubernetes.

Gut zu wissen

Die Abkürzung K8s kommt durch das Ersetzen der acht Buchstaben »ubernete« mit der Zahl 8.

Hinweis

Das Buch wurde über Kubernetes in der Version v1.27 geschrieben. Wenn in Ihrem Unternehmen Cluster einer älteren Version eingesetzt werden, dann sind einige Features vermutlich nicht vorhanden. Bei neuen Features weise ich im entsprechenden Kapitel gesondert darauf hin.

Wenn Sie sich unsicher sind, ob ein Feature bei Ihnen im Unternehmen genutzt werden kann, dann können Sie in der Kubernetes-Dokumentation auf *https://kubernetes.io/* nachsehen oder bei Ihrem Administrator anfragen.

2.1 Grundlagen und Konzepte: Warum überhaupt Container-Cluster?

Um Kubernetes besser zu verstehen, nehme ich Sie mit in die Vergangenheit und die Entstehung. Am 7. Juni 2014 erblickte Kubernetes das Licht der Welt. Zumindest der öffentlichen Welt, denn an diesem Tag wurde der erste Commit auf GitHub publiziert. Die Idee für eine Container-Management-Plattform war jedoch nicht neu. Sie entstand bei Google schon in den 2000er Jahren, denn schon damals mussten Entwickler dort mehrere hunderttausende Container betreiben.

Bei der Masse an Containern brauchte Google ein System, das die Verwaltung und den Betrieb von vielen Servern vereinfacht. Doch damals gab es noch keinen großen Markt dafür, und die Entwickler von Google bauten sich ihre Lösung selbst. Borg war geboren.

> *Borg simplifies life for its users by offering a declarative job specification language, name service integration, real-time job monitoring, and tools to analyze and simulate system behavior.*

Google setzte Borg ein, um eine Vielzahl von Herausforderungen im Zusammenhang mit dem Management großer Cluster von Maschinen zu bewältigen. Probleme, die durch Borg gelöst wurden, waren beispielsweise:

- **Ressourcenmanagement:** Borg automatisierte das Scheduling, Starten, Stoppen, Neustarten und Überwachen von Containern. Das ermöglichte den Entwicklern, sich auf ihre Entwicklungsarbeit zu konzentrieren, statt sich mit der Verwaltung von Ressourcen zu beschäftigen.
- **Effizienz und Auslastung:** Durch Techniken wie Überbuchung (*Overcommitment*) ermöglichte Borg eine hohe Auslastung der verfügbaren Ressourcen. Das sparte Google hohe Rechenzentrumskosten.
- **Fehlerbehandlung:** Borg bot Laufzeitfunktionen und Scheduling-Regeln, die die Zeit zur Fehlerbehebung verringerten.

Gut zu wissen

Die Einführung von Borg war ein entscheidender Schritt für Google, um seine Infrastruktur effizient zu verwalten. Wo sie früher die Server selbst überwachten und managten, konnte das durch Borg übernommen werden.

Sie werden auch im Laufe des Buchs sehen, dass Kubernetes bei Hardwarefehlern eines Servers all Ihre Anwendungen automatisch auf einen funktionierenden Server umzieht. Sie sparen sich dadurch Zeit und verringern gleichzeitig die Ausfallzeit.

Auch heute noch erwarten wir uns von einem Management-System genau diese Vorteile. Die Welt hat sich seitdem weiterentwickelt, immer mehr Unternehmen setzten auf Container, und auch Borg musste sich weiterentwickeln.

Kubernetes sollte eine Neuentwicklung für das bereits bestehende Container-Management-Tool werden. Die jahrelange Erfahrung mit Borg sollte in ein neues Design fließen. Teile, die funktionierten, wurden übernommen, und andere Teile wurden optimiert. Der wohl größte Unterschied zu Borg ist das neue Lizenzmodell. Die Entwickler von Google entschieden sich für eine Open-Source-Modell und spendeten Kubernetes in der Version 1.0 an die Cloud Native Computing Foundation. Das macht Kubernetes zu einem offenen und unabhängigen System, das vielleicht auch gerade dadurch aktuell sehr beliebt ist.

Gut zu wissen

Die Cloud Native Computing Foundation ist Teil von The Linux Foundation, die sich selbst auf der Webseite folgendermaßen vorstellt:

The Linux Foundation provides a neutral, trusted hub for developers and organizations to code, manage, and scale open technology projects and ecosystems.

Eine Stiftung als Gesellschaft für eine Open-Source-Technologie steigert aus meiner Sicht das Vertrauen und die Unabhängigkeit von Kubernetes.

Mehr zu der Geschichte von Borg und der Geburt von Kubernetes finden Sie unter diesen beiden Links:

- *https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes/*
- *https://research.google/pubs/large-scale-cluster-management-at-google-with-borg/*

2.1.1 Wieso überhaupt Container?

Vielleicht haben Sie selbst schon Container entwickelt und betrieben. Docker ist aktuell der bekannteste Vertreter für Container, und meist wird es als Synonym verwendet. So wie Tempo das Taschentuch ist, so ist Docker der Container. Docker hat das Konzept von Containern nicht erfunden, doch sehr viel dafür gemacht, dass es heute so weit verbreitet ist. Das ist auch verständlich, denn Container

- sind leichtgewichtig,
- einfach zu nutzen und
- laufen quasi auf jedem Server, der eine Runtime hat.

Zusätzlich dazu sind die Images einfach zu transportieren, und es steckt alles drin, was Ihre Anwendung braucht. Es gibt nicht mehr das Problem »Auf meinem Rechner läuft es aber!«.

Vergleichen wir Container mit virtuellen Maschinen, dann liegt der größte Vorteil auf der Hand: Sie brauchen bei einem Container kein vollständiges Betriebssystem zu installieren.

Vielleicht kennen Sie die Evolution der Virtualisierung schon, die Sie in Abbildung 2.1 sehen.

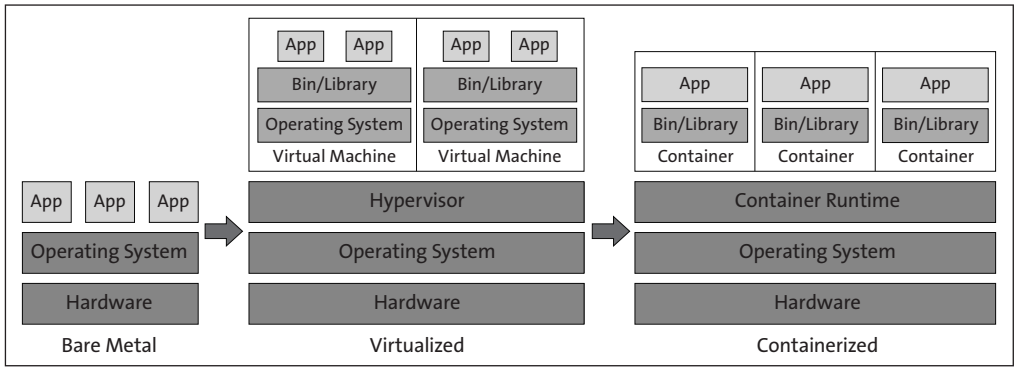

Abbildung 2.1 Evolution der Virtualisierung

Dort sehen Sie, wie sich die Bereitstellung von Anwendungen im Laufe der Zeit von Bare Metal zur Virtualisierung entwickelt hat. (Das soll nicht heißen, dass Container die virtuellen Maschinen ablösen. Aber sie laufen ihnen in vielen Einsatzszenarien doch den Rang ab.)

Doch wieso hat sich das so entwickelt? Nehmen wir dazu ein stark vereinfachtes Beispiel:

Stellen Sie sich ein Rechenzentrum vor. Dort gibt es Racks, die mehrere Bare-Metal-Server enthalten können. Ein Rack hat eine maximale Kapazität an Servern, die es aufnehmen kann, und das Rechenzentrum hat eine maximale Kapazität an Racks, die es aufnehmen kann. Wenn Sie jetzt an einen regionalen Webshop denken, der auf einem der Server läuft, dann ist dort abends mehr los als mitten in der Nacht. Der Server hat also nachts nichts zu tun und heizt unnötig das Rechenzentrum. Wenn Sie nur solche Server haben, dann haben Sie im gesamten Rechenzentrum eine sehr schlechte Auslastung und dadurch hohe Kosten.

Hinzu kommt, dass Sie den Server auf die Lastspitze auslegen müssen, damit auch in Hochzeiten jeder Kunde seine Produkte kaufen kann. Sie haben also generell einen überdimensionierten Server. Ein weiterer Punkt ist die Abhängigkeit vom Betriebssystem und der darunterliegenden Hardware. Sie haben ein einziges Betriebssystem

installiert mit den Treibern für die Hardware des Servers. Sie können nicht einfach einen Klon davon machen und auf einem anderen Server installieren, was wiederum das Backup und Recovery schwieriger macht.

Hinweis

Natürlich können Sie auf einem Server auch mehrere Anwendungen installieren. Wenn zu den Zeiten, wo auf dem Webshop gerade nichts los ist, eine andere Anwendung beispielswiese Batch-Jobs ausführt und Abrechnungen macht, dann erhöhen Sie auch die Auslastung, doch Virtualisierung bringt noch mehr mit.

Wie können Sie die Auslastung eines Servers erhöhen und die Schwierigkeiten des Bare-Metal-Servers überwinden? Wenn Sie dem Server ermöglichen, virtuelle Maschinen auszuführen, dann können Sie auf einer Bare-Metal-Instanz mehrere virtuelle Server laufen lassen. Durch mehr Instanzen schaffen Sie eine bessere Auslastung und verteilen sogar die Kosten des Servers auf mehrere virtuelle Server. Doch nicht nur die Auslastung wird besser:

- Sie sind unabhängiger von der eigentlichen Hardware und können Ihre virtuelle Maschine (VM) ohne viel Aufwand auch auf anderen Servern laufen lassen.
- Mit VMs können Sie sehr einfach Backup- und Recovery-Prozesse aufsetzen.
- Sie können durch Golden Images Standards setzen, die auch einfach genutzt werden können.

Virtuelle Maschine sind also eine Optimierung des Bare-Metal-Servers mit jeweils einem eigenen Betriebssystem. Sie verhalten sich wie richtige Server – müssen beim Startup alles hochfahren und haben weiterhin den Overhead eines normalen Servers. Da stellt sich die Frage: Geht es noch einfacher und kleiner? Die Antwort ist Containerisierung.

Gut zu wissen

Unternehmen, die ihre Kubernetes-Cluster in der Cloud betreiben, bauen sogar meist ihre Cluster auf virtuellen Maschinen auf. Das ist sinnvoll, denn bei AWS fangen die Bare-Metal-Instanzen erst bei 48 CPU-Kernen und 384 GB Arbeitsspeicher an. Damit könnten Sie locker alle Container von kleineren Clustern auf einer einzigen Instanz betreiben, aber das wäre bei einem Hardwarefehler fatal.

Für Redundanz und Skalierbarkeit gilt daher: besser kleinere Instanzen, aber dafür mehr davon.

Nehmen wir den Webshop und verpacken ihn und alles, was wir für den Betrieb brauchen, in einen Container. Wir lösen die Anwendung also von der virtuellen Maschine und können den Webshop unabhängig von ihr benutzen. Alles, was Sie für einen

Container brauchen, ist eine Runtime, die entweder direkt auf der Bare-Metal-Instanz installiert ist oder sogar in einer virtuellen Maschine. Damit nutzen Sie die Vorteile der Container:

- Container nutzen im Vergleich zu virtuellen Maschinen deutlich weniger Systemressourcen, da sie ohne vollständiges Betriebssystem auskommen.
- Dank der Containerisierung lassen sich Anwendungen mühelos über verschiedene Betriebssysteme und Hardwareumgebungen hinweg einsetzen.
- Durch den Einsatz von Containern lassen sich Anwendungen schneller ausrollen, aktualisieren und skalieren.
- Container beschleunigen auch den Entwicklungsprozess, und durch die Portabilität der Images können sie auf jedem Entwicklerrechner ausgeführt werden.

Container haben also einige Vorteile gegenüber den virtuellen Maschinen. Sie lösen jedoch weder die virtuellen Maschinen noch die Bare-Metal-Server ab. Alles davon hat seine Daseinsberechtigung und einen entsprechenden Anwendungsfall. Das Beispiel zeigt jedoch, wieso Anwendungen heutzutage fast nur noch im Container entwickelt werden.

Gut zu wissen

Der Container ist im Vergleich zu den virtuellen Maschinen nochmals besser für die Auslastung Ihrer Server. Denn je kleiner die Einheit ist, desto einfacher findet sich noch eine Lücke auf einem Server.

Nehmen Sie wie in Abbildung 2.2 beispielsweise ein Glas voll mit Murmeln. Zwischen den Murmeln ist noch genügend Luft, um kleine Perlen einzufüllen, und danach ist zwischen den Perlen noch genug Luft, um feinen Sand einzufüllen.

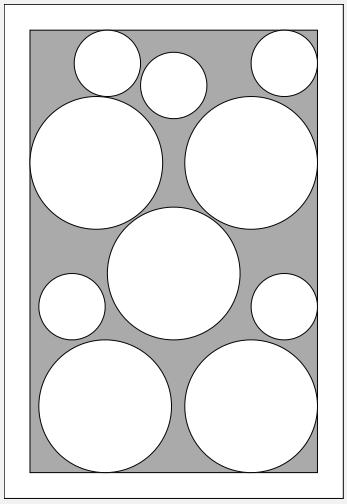

Abbildung 2.2 Glas mit Murmeln und Perlen

Wenn wir uns eine moderne Anwendung ansehen, dann ist die Handhabung in Containern deutlich einfacher. Schon der Startup ist deutlich schneller, und das verändert wieder die Art, wie Skalierung funktioniert. Früher hat der physische Server ein Upgrade der CPU oder des Arbeitsspeichers erhalten, damit die monolithische Anwendung mehr Power hatte.

Heute muss einfach nur ein weiterer Container mit der gleichen Anwendung gestartet werden, und innerhalb von wenigen Minuten wird die Last schon auf den neuen Container verteilt.

Ist kein Platz mehr auf dem Server, dann startet wünschenswerterweise wie von Zauberhand ein neuer Server in der Cloud, und der Container wird dort deployt. Der Trend geht von der vertikalen Skalierung zur horizontalen Skalierung – doch so einfach, wie sich das vielleicht manchmal anhört: Hinter so einem Cluster-Setup steckt viel Know-how und Arbeit.

Gut zu wissen

Das Schöne an horizontaler Skalierung ist, dass Sie sich nicht mehr auf einen großen Server als *Single Point of Failure* verlassen müssen. Wenn die Last auf viele kleinere Server verteilt ist, dann können Sie einen Ausfall viel einfacher ausgleichen.

Aktuell gehen die Technologien und die Softwarearchitektur Hand in Hand. Neue Anwendungen werden meist nur noch in einer Microservice-Architektur entwickelt, die Kommunikation muss am besten asynchron und eventbasiert sein. Unternehmen wollen ihre Workloads in Clouds auslagern und dank automatischer und bedarfsgerechter Skalierung nur noch das bezahlen, was sie wirklich brauchen.

2.1.2 Warum brauchen Sie ein Container-Management-Tool?

Viele Microservices und horizontale Skalierung bringen neue Herausforderungen mit sich. Auf einmal gibt es Hunderte oder Tausende von Containern, die gleichzeitig betrieben und überwacht werden müssen.

Ich erinnere mich gerne zurück an die Zeit, als ich zum ersten Mal mit Containern in Berührung gekommen bin. 2017 arbeitete ich in einem Unternehmen, das sich gerade in einer großen Transformation befand. Es hatte gerade die Entscheidung getroffen, dass jede Software, egal ob Legacy-Anwendung oder Neuentwicklung, in die Cloud von Amazon migriert werden sollte. In diesem Zuge wurde ein Programm aufgesetzt, das eine alte Vertriebsplattform neu entwickeln sollte, und alle neuen Microservices sollten containerisiert werden und auf Kubernetes laufen.

Das war gerade mal zwei Jahre nach Veröffentlichung der Kubernetes-Version 1.0. Die neue Container-Welt funktioniert vollkommen anders als die Anwendungen, die vor Jahrzehnten entwickelt wurden. Wenn früher eine Webanwendung entwickelt wurde,

dann hatte diese meist ein monolithisches Design und lief als virtuelle Maschine auf einem Server im Rechenzentrum. Diese Anwendungen waren auch häufig auf eine bestimmte Anzahl von Usern ausgelegt. Skalierung nach Bedarf war nicht einfach möglich. So auch in diesem Unternehmen.

Wenn die alte Vertriebsplattform mehr Traffic erhalten hat als erwartet, weil es beispielsweise eine Weihnachtsaktion gab, dann musste die Anwendung mehr CPU und Memory erhalten. Es konnte also nur vertikal skaliert werden. Meist ging das einher mit wochenlanger Vorbereitung und Planung, bei der Mitarbeiter sich nur um das Kapazitätsmanagement kümmerten. In der restlichen Zeit lief diese Anwendung auf 30 % Last und heizte unnötig das Rechenzentrum.

Die Architektur von neuen Applikationen geht weg von monolithischen Designs hin zu Microservice-Architekturen: kleinere unabhängige Services, die am besten asynchron kommunizieren und dadurch lose gekoppelt sind. So hätten wir beim Beispiel einer Webanwendung einen Webserver wie *Nginx*, der bei Bedarf einfach skaliert wird. Ob die Anfrage des Nutzers vom einen oder vom anderen Nginx beantwortet wird, ist sowohl dem Webserver als auch dem Nutzer egal. Hauptsache, die Antwort ist die gleiche. Durch die lose Kopplung können wir genau diesen Teil des Systems skalieren, der gerade einen Engpass in der Kapazität hat.

Es ist also nicht so, dass wir die monolithischen Anwendungen in Container packen und auf Kubernetes deployen müssen, um dann in der schönen neuen Welt angekommen zu sein. Es steckt ein vollkommen neues Konzept dahinter und bedarf einer neuen und modernen Microservice-Architektur, die wiederum ein Umdenken im Unternehmen braucht. Denn die Microservice-Architektur führt dazu, dass es auf einmal nicht nur einige wenige monolithische Anwendungen gibt, sondern mehrere kleine Anwendungen. Die Anzahl in größeren Projekten wird schnell zweistellig und in Unternehmen gerne mal drei- oder vierstellig. Dies verändert auch die Herausforderungen im Betrieb.

Mit kleineren Services kommt auch häufig die Chance, durch *Continuous Integration und Continuous Delivery (CI/CD)* Updates zu liefern. Das hat den Vorteil, dass

- Entwickler einfacher und schneller deployen können,
- mehr Verantwortung bei den Entwicklern liegt,
- der Betrieb entlastet wird und
- eine höhere Qualität wichtig wird.

Aber auch den Nachteil, dass

- jede Veränderung die Gefahr von Fehlern birgt,
- Abhängigkeiten zu anderen Komponenten vergessen werden können und
- Release-Prozesse ignoriert werden.

Damit nicht im Minutentakt die Tickets im IT-Betrieb geöffnet werden und das Telefon nicht mehr stillsteht, muss auch der Grad der Automatisierung deutlich erhöht werden.

Ein weiterer Aspekt ist die Veränderung der Prozesse und manchmal auch der gesamten Organisation. Es verändert sich meist nicht nur die Technologie, sondern kulturelle Veränderungen durch die Prinzipien von *DevOps* oder Prozessoptimierungen durch *Lean* und *ITIL* gehen mit der Technologie Hand in Hand. Sie als Entwicklerin oder Entwickler stehen zwischen all den Veränderungen und sollen mal ganz schnell nebenbei hervorragende Software entwickeln, die stabil im Betrieb ist und den Endkunden glücklich macht.

In dem ganzen Chaos kommt Kubernetes ins Spiel. Durch das Container-Management-Tool können Sie Ihre Container automatisiert überwachen und bei Bedarf skalieren, neu starten oder beenden lassen. Es fügt sich als Plattform sehr gut in moderne Prozesse ein und gibt Ihnen als Entwickler mehr Eigenverantwortung. Auch bei Releases sorgt Kubernetes dafür, dass kein Ausfall beim Nutzer zu spüren ist, und wir werden uns noch genauer ansehen, wieso Kubernetes den Betrieb von Microservices vereinfacht.

2.1.3 Von Pets und Cattle

Sie haben schon gelesen, dass es im Unternehmen ein Umdenken geben muss. Vielleicht kennen Sie den klassischen Vergleich von Pets und Cattle auch schon. Pets sind Haustiere, und Cattle ist das Vieh des Bauern. In der Welt von monolithischen Anwendungen werden die Server meist wie Haustiere behandelt. Ein Haustier wird gefüttert, gepflegt und geliebt. Es hat einen Namen und gehört zur Familie, und es kann nicht einfach durch ein anderes Tier ersetzt werden.

Auch bei der Infrastruktur der monolithischen Anwendungen waren die Server schwer zu ersetzen. Wenn die Infrastruktur aus dem Support läuft, dann sind Migrationen notwendig, die mit großen Risiken verbunden sind. Es soll also möglichst vermieden werden, dass es dem Server schlecht geht, und an einen neuen Server ist gar nicht zu denken. Ersatzteile, wie Festplatten und Netzteile, liegen im Tresor und warten auf ihren Einsatz, damit die Anwendungen weiterhin laufen.

Schauen wir uns als Gegenbeispiel das Vieh eines Bauern an, dann können wir eine ganz andere Art von Liebe beobachten. Auch der Bauer pflegt sein Vieh, doch die Kühe haben zur Identifikation statt Namen eher Nummern am Ohr. Sie haben eine klare Aufgabe, geben Milch und Fleisch. Nach einer gewissen Zeit, wenn ein Tier zu alt oder krank wird, dann wird es einfach durch ein anderes ersetzt.

Hinweis

Das Standardbeispiel mit Pets und Cattle ist vielleicht etwas makaber. Eine schöne Alternative ist der Vergleich zwischen Wildblumen und Bonsaibäumen.

Ein Bonsai erfordert kontinuierliche Pflege und Aufmerksamkeit. Sie beobachten regelmäßig seine Form, Größe, den Bodentyp und die Zufuhr von Dünger. Die Verlagerung eines Bonsais in eine neue Umgebung oder die Änderung seiner Pflegebedingungen können erhebliche Auswirkungen auf sein Wohlergehen und Wachstum haben.

Wildblumen hingegen sind von Natur aus widerstandsfähig. Sie wachsen, wo die Bedingungen es zulassen, ohne spezifische Anforderungen an den Standort oder die Umgebung zu stellen. Wenn ein Bereich nicht mehr geeignet ist, können Sie sie einfach an einem neuen Ort aussäen, ohne dass die vorherige Position oder spezielle Bedingungen berücksichtigt werden müssen.

Seit es Cloud Computing gibt, verändert sich die Handhabung von Infrastruktur immer mehr Richtung Cattle. Wenn ein Server nicht mehr funktioniert, dann wird ein neuer aufgebaut, der die Aufgabe übernimmt. Aufbau und Migration der Anwendungen passiert automatisiert.

Kubernetes kann diese Aufgaben übernehmen. So müssen Sie sich keine Gedanken über die Infrastruktur machen. Die Vorteile liegen auf der Hand. Fehler der Infrastruktur werden automatisiert gelöst, und die Infrastruktur skaliert nach Bedarf. Dadurch sparen Unternehmen Geld, weil sie nur noch so viel Rechenleistung bezahlen, wie sie auch wirklich brauchen, und durch die Automatisierung wird der ein oder andere Rufbereitschaftseinsatz verhindert.

Doch nicht nur die Server werden als Cattle behandelt. Auch die Applikationen, die in Containern laufen, sind keine Haustiere mehr. Kennen Sie schon den *Chaos Monkey*? Das ist ein Tool, das von Netflix entwickelt wurde, um die Stabilität von Produktionssystemen zu prüfen. Stellen Sie sich vor, in Ihr Rechenzentrum ist ein Affe eingebrochen. Er beißt zufällig Kabel durch und haut mit einem Hammer auf die Server ein. Würde Ihre Anwendung das überleben?

Sie müssen jetzt nicht direkt den Chaos Monkey in Ihrem Unternehmen einführen, doch der Gedanke dahinter ist gut. Stellen Sie sich einfach die Frage: Würde es der Nutzer merken, wenn eine Komponente oder ein Container ausfällt? Ist die Antwort ja, dann gibt es auf jeden Fall noch Verbesserungspotenzial.

2.1.4 Stateless- und Stateful-Applikationen

Um eine Applikation für Kubernetes zu entwickeln, ist eine Kernfrage wichtig: Ist Ihre Anwendung *stateless* oder *stateful*? Also muss sich Ihre Anwendung einen State merken? Doch was unterscheidet diese beiden Konzepte?

Stellen Sie sich vor, Sie sind in Italien und sitzen in einem Café in einem kleinen Örtchen in der Nähe von Venedig. Sie haben direkten Blick auf die Adria und bestellen einen Kaffee. Der Kellner bringt Ihnen Ihren Kaffee, und auf der Untertasse liegen Milch und Zucker. Dabei trinken Sie Ihren Kaffee schwarz.

Das ist ein gutes Beispiel für *stateless*. Das Café selbst speichert keine Informationen über Sie als Kunden und kennt auch nicht Ihre vorherigen Bestellungen. Wenn Sie etwas bestellen, dann bekommen Sie das gleiche wie jeder andere Kunde auch. Jeder Auftrag wird isoliert behandelt, ohne dass eine vorherige Historie berücksichtigt wird. Es ist also stateless, da es keinen dauerhaften Zustand oder Informationen speichert.

Nun, lassen Sie uns das Konzept *stateful* betrachten. Wenn ich mit meinem Freund Fabian in sein Lieblingscafé gehe, dann nickt er dem Kellner nur zu, und wir bekommen beide einen schwarzen Kaffee. Keine Milch und keinen Zucker. Der Kellner kennt uns einfach.

Das ist, wie wenn Sie Mitglied in einem Fitnessstudio sind und dort Ihre persönlichen Daten und Trainingsfortschritte auf Ihrem Mitgliedskonto speichern. Sie gehen zur Beinpresse, stecken Ihre Karte in das Gerät, und es wird Ihnen das passende Gewicht zu Ihrem Trainingsfortschritt vorgeschlagen. Hier wird der Zustand gespeichert und fortlaufend aktualisiert, um eine personalisierte Erfahrung zu bieten. Das Fitnessstudio oder Ihr Stammcafé ist also stateful, da es Informationen über Sie speichert und nutzt, um Ihre Trainingsroutine zu verbessern oder Ihnen direkt Ihr Lieblingsgetränk zu bringen. Der gespeicherte Zustand bringt Annehmlichkeiten, aber auch mehr Verantwortung mit sich. Sie müssen sich Gedanken darüber machen, wie der Status gespeichert wird. Was machen Sie beispielsweise, wenn der Kellner, der uns kennt, krank ist?

States bringen Herausforderungen mit sich:

- Daten müssen konsistent über alle Instanzen gehalten werden. (Jeder Kellner muss uns kennen.)
- Horizontale Skalierung ist schwerer. (Ein neuer Kellner muss erstmal eingearbeitet werden.)
- Sie müssen sich um Backup und Recovery Gedanken machen. (Wie werden die Daten wiederhergestellt, wenn der Kellner ausfällt?)

Übertragen auf die Welt der IT können Sie eine einfache Website mit einem Onlineshop vergleichen. Wenn Sie sich die Website einer kleinen Schreinerei aus dem Nachbardorf vorstellen, dann finden Sie dort Bilder von Projekten, Informationen über die Erreichbarkeit und vielleicht ein Kontaktformular. Das sind alles Daten, die jedem Nutzer beim Aufruf der Website angezeigt werden. Es müssen keine Daten vorgehal-

ten oder gespeichert werden, und selbst beim Kontaktformular wird einfach eine E-Mail an die Geschäftsführerin gesendet. Die Website ist stateless.

Der Onlineshop eines großen Möbelhauses dagegen bietet Funktionen an, die einen State benötigen. Denken Sie beispielsweise an den Warenkorb. Dieser enthält alle Ihre Produkte, die Sie einkaufen möchten. Auch die Bestellhistorie und die Rechnungen sind Daten, die über Sie gespeichert werden müssen. Dank der Daten kann der Onlineshop Ihnen auch Vorschläge unterbreiten, wie »Kunden, die einen Tisch kauften, kauften auch einen Stuhl« oder Kategorien, passend auf Sie abgestimmt.

In einfacheren Worten, stateless bedeutet, dass jede Interaktion unabhängig ist und keine Informationen über vorherige Interaktionen enthält. Stateful hingegen bedeutet, dass Informationen über vergangene Interaktionen gespeichert und genutzt werden, um eine kontinuierliche Erfahrung zu ermöglichen. Sie merken jedoch an den Beispielen, dass es in den meisten Anwendungen States gibt, die es zu speichern gilt.

Dass States gespeichert werden müssen, können wir meistens nicht steuern. Was wir jedoch steuern können, ist die Art, wie wir unsere Anwendungen schneiden. Möglichst viele davon sollten stateless sein, denn diese sind deutlich einfacher zu handhaben.

Stateless-Anwendungen sind

- einfacher zu skalieren und performanter,
- einfacher zu deployen sowie
- besser zu verwalten und zu debuggen.

Gut zu wissen

Je nachdem, ob eine Anwendung Daten speichert oder eben nicht, muss diese in Kubernetes unterschiedlich behandelt werden.

Eine Datenbank ist stateful, und wird diese in Kubernetes betrieben, dann kann Kubernetes sie nicht einfach terminieren oder neu aufbauen. Für diesen Anwendungsfall gibt es ein eigenes Objekt, das genau diese Eigenschaften mit sich bringt, um Stateful-Anwendungen zu betreiben. Sie werden es in Abschnitt 6.1 kennenlernen.

2.1.5 Separation of Concerns

Separation of Concerns, oder auf Deutsch Trennung der Verantwortlichkeit, ist ein Designprinzip in der Softwareentwicklung. Es zielt darauf ab, komplexe Systeme in mehrere Komponenten oder Module aufzuteilen. Jedes Modul hat eine klar defi-

nierte und begrenzte Verantwortung oder Aufgabe. Durch die Aufteilung ist die Wartung, die Weiterentwicklung und die Verständlichkeit der Software deutlich besser.

Ein klassisches Beispiel ist die Drei-Schichten-Architektur, wie in Abbildung 2.3, die Sie vielleicht schon kennen. Wir nehmen mal an, Sie entwickeln einen Onlineshop. Hier könnten Sie eine Trennung nach Frontend, Backend und Datenbank machen. Das Frontend ist für das Ausliefern der HTML-, CSS- und JavaScript-Dateien verantwortlich. Das ist am Ende das, was der Kunde sieht. Er kann sich Produkte in den Warenkorb legen, diese am Ende auschecken und kaufen.

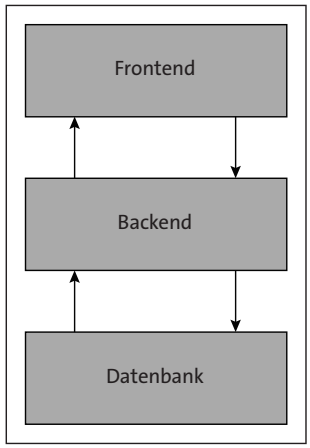

Abbildung 2.3 Einfache Drei-Schichten-Architektur

Das Backend ist die Anwendung, die im Hintergrund läuft und Anfragen des Frontends verarbeitet. Wenn der Kunde seine Kreditkartendaten eingegeben hat, dann kann sich das Backend um die eigentliche Abrechnung kümmern.

Zu guter Letzt braucht es eine Persistenz, die mit einer Datenbank abgebildet wird. Diese wird vom Backend genutzt, um Daten zu speichern oder abzufragen. Dort wird zum Beispiel die Bestellhistorie des Kunden abgelegt. Zwischen den einzelnen Komponenten gibt es API-Schnittstellen, die die Kommunikation dazwischen ermöglichen.

Gut zu wissen

Die Drei-Schichten-Architektur ist eine Möglichkeit, doch es gibt auch andere. Einer meiner Kunden hat mehrere Softwareprodukte im Einsatz. Manche davon sind Legacy-Anwendungen im Rechenzentrum und andere SaaS-Anwendungen in der Cloud. Um Daten zwischen den Systemen auszutauschen, haben wir Konnektoren entwickelt – für jede Datenstrecke einen eigenen Konnektor nach dem Prinzip ETL (*Extract, Transform, Load*).

Durch diese Trennung können Entwickler, die an der Benutzeroberfläche arbeiten, dies tun, ohne sich Gedanken über die Funktionsweise der Geschäftslogik oder den Datenbankzugriff machen zu müssen.

Durch die API ist festgelegt, wie mit dem Backend kommuniziert werden kann und welche Funktionen es anbietet. Gleichzeitig können die Backend-Entwickler an der Geschäftslogik arbeiten, ohne an das Frontend denken zu müssen. Diese Trennung erleichtert die Wartung und Erweiterung der Anwendung erheblich, denn Sie wissen selbst, dass es für die jeweiligen Komponenten unterschiedliche Programmiersprachen gibt und meist auch unterschiedliche Entwickler oder sogar ganze Teams.

Die Aufteilung sieht dann häufig aus wie in Abbildung 2.4, und die Entwickler können sich auf bestimmte Module fokussieren.

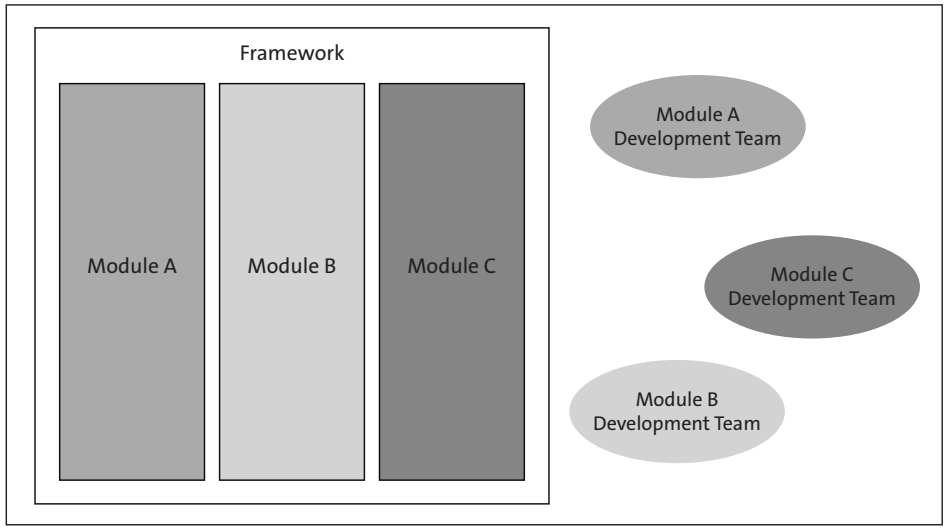

Abbildung 2.4 Zuständigkeit von Entwicklern

In der Welt von Containern wird oft versucht, die Komponenten in atomare Einheiten aufzuteilen, um die Separation of Concerns so stark wie möglich zu machen. So ist es nicht selten der Fall, dass das Backend in mehrere Microservices aufgeteilt wird. Sie haben dann einen Microservice für den Warenkorb, die Abrechnung, die Bestellung und so weiter. Je nach System ist sogar eine noch granularere Einteilung sinnvoll. Je kleinteiliger das System ist, desto größer kann der Aufwand werden, es zu pflegen. Der Overhead wird dann größer, weswegen es wichtig ist, zu prüfen, wo der Schnitt gemacht werden muss.

Gut zu wissen

Der Begriff *atomar* leitet sich aus dem griechischen Wort *atomos* ab und bedeutet »unteilbar«. Wenn ich hier von atomaren Komponenten spreche (die Sie in Con-

tainern ausliefern), dann bedeutet das immer, dass diese nicht in kleinere Einheiten zerlegt werden können. Sie sind Teil einer größeren Anwendung, doch die Trennung nach Aufgaben bzw. Verantwortungen hilft in vielen Bereichen.

So wie Sie als Entwickler einzelne Funktionen in Ihrem Code klein halten, so wollen Sie auch die Komponenten klein halten. Sie sollen eine Aufgabe haben und nicht mehr.

In Entwicklungsprojekten sehe ich es häufig, dass im Vorfeld nicht immer alles betrachtet werden kann. Seien Sie auch im späteren Projektverlauf mutig, die Komponenten nachträglich zu trennen oder zusammenzulegen. Auch ich bin immer auf der Suche nach dem Sweet Spot, um weder in das eine Extrem (zu kleinteilig) noch in das andere (große Komponenten) zu verfallen. Welche Vorteile können Sie durch eine gute Aufteilung erreichen?

- **Granulare Skalierbarkeit:** Mit atomaren Containern können Sie genau die Teile Ihrer Anwendung skalieren, die benötigt werden. Wenn beispielsweise die Zahl der Webserver erhöht werden muss, dann können Sie diese skalieren, ohne direkt die Datenbank mitskalieren zu müssen.
- **Unabhängige Aktualisierungen:** Atomare Container ermöglichen es, Teile Ihrer Anwendung unabhängig voneinander zu aktualisieren. Dies minimiert Ausfallzeiten und vereinfacht das Deployment.
- **Bessere Ressourcennutzung:** Da jede Komponente in einem eigenen Container isoliert ist, können Sie die Ressourcen bedarfsgerecht verteilen. In der Kombination mit der einfachen Skalierung sind Sie deutlich flexibler, sparen Ressourcen und dadurch Geld.
- **Erhöhte Sicherheit und Stabilität:** Die Trennung erhöht die Sicherheit, denn falls ein Container kompromittiert wird, sind die anderen Container nicht unbedingt gefährdet. Zudem führt ein Fehler in einem Container nicht direkt zum Ausfall des ganzen Systems.
- **Einfache Wartung und Weiterentwicklung:** Die Trennung in atomare Komponenten erleichtert Ihnen auch die Weiterentwicklung und Wartung. Sie müssen deutlich weniger *Reverse Engineering* betreiben.

Wenn Sie keine Trennung haben, dann ist das genauso schlecht, wie wenn Sie zu kleinteilig unterwegs sind. Die Herausforderung ist, die richtige Aufteilung für Ihren Anwendungsfall zu finden. Wenn Sie mit einer Hypothese starten, dann werden Sie im Laufe der Zeit herausfinden, ob Sie noch weiter einteilen müssen oder wieder Komponenten zusammenlegen.

Trauen Sie sich einfach, loszulegen.

2.2 Kubernetes, das Tool der Wahl

Sie haben jetzt erfahren, wozu und warum ein Container-Management-Tool gebraucht wird, und haben die wichtigsten übergreifenden Konzepte kennengelernt. Sie wissen auch, wie Kubernetes entstanden ist, doch warum ist Kubernetes das Tool der Wahl?

Zu Beginn möchte ich auf die Beweggründe eingehen, aus denen Unternehmen überhaupt auf Kubernetes setzen wollen, denn sind wir mal ehrlich: Ein Unternehmen optimiert nicht seine IT, damit es die modernsten Software-Tools einsetzt, sondern es steckt immer ein unternehmerischer Zweck dahinter.

Danach führe ich Sie kurz durch die Argumente, mit denen Kubernetes in den Markt geht, und gebe Ihnen einen Einblick in meine Erfahrung, die ich dazu in der Realität gemacht habe. Hält Kubernetes wirklich, was es verspricht?

Zum Abschluss des Kapitels werde ich noch mal klar hervorheben, für welche Unternehmen Kubernetes sinnvoll ist und für welche Unternehmen eben nicht. Denn Kubernetes ist wahrlich nicht das Allheilmittel für jedes IT-Problem.

2.2.1 Wieso Unternehmen auf Kubernetes setzen wollen

Die erste Frage, die ich meinen Kunden stelle, wenn sie Kubernetes einführen möchten, ist: »Welches Ziel wollen Sie damit erreichen?« Die Antworten dazu können vielfältig sein. In meiner Erfahrung lässt es sich immer auf folgende drei Punkte herunterbrechen:

- schnellere Time-to-Market
- Kosten sparen durch optimierte Prozesse
- neue Märkte erschließen durch neue Software

Doch am Ende läuft es immer auf klare, messbare Fakten hinaus, und das ist meist: mehr Geld verdienen oder weniger Geld ausgeben. Wie ist es bei Ihnen im Unternehmen? Baut Ihr Unternehmen schön länger auf Kubernetes oder gehören Sie zu den Ersten im Entwicklungsteam, die Kubernetes einsetzen sollen? Kennen Sie die Ziele dahinter?

Sie haben das sicherlich auch schon selbst erlebt. Sie entwickeln an einem Prototyp, und vom einen auf den anderen Tag wird entschieden: Das Projekt wird gestoppt. Nur wenn Sie als Entwickler oder Entwicklerin auch im Vorfeld die Ziele des Unternehmens kennen, können Sie schon während der Entwicklung darauf achten, dass das Projekt ein Erfolg wird. Sie tragen einen großen Teil zum Erfolg Ihres Unternehmens bei und können durch die Klarheit in den Zielen auch Ihre Aufgaben deutlich besser priorisieren.

Hinweis

Die Rollbacks durch ein Deployment sind ein großartiges Werkzeug, um bei Problemen schnell reagieren zu können. Produktiv nutze ich diese jedoch fast nie. Das Problem dabei ist die Nachvollziehbarkeit von Änderungen, denn wenn Sie ein Rollback durchführen, dann sind Ihre Manifeste im Git-Repository nicht mehr aktuell, und andere Teammitglieder können die Änderungen schwerer nachvollziehen.

Im Repo haben Sie die Historie und sehen auch direkt die Änderungen des Codes. Passende CI/CD-Pipelines werden Ihnen auch ein schnelles Rollback ermöglichen, und Sie haben alle Änderungen an Ihrem produktiven System getrackt.

3.4 ConfigMaps und Secrets

Wenn Sie Container entwickeln und bauen, gehört mit wenigen Ausnahmen alles in den Container hinein, was Ihre Anwendung benötigt. Das eine sind Secrets wie Passwörter oder Zertifikate, und das andere sind Konfigurationen. Denn Sie wollen nicht für jede Umgebung ein eigenes Image bauen, sondern Sie wollen einen Container in unterschiedlichen Umgebungen deployen und wie in Abbildung 3.26 durch umgebungsspezifische Konfiguration konfigurieren können.

Kubernetes bietet dafür die *ConfigMap* und das *Secret* an. Die beiden Objekte sind wie Notizbücher, die Sie auf Ihrem Schreibtisch liegen haben. Jeder aus Ihrem Unternehmen kann einen Blick hineinwerfen, wobei das Notizbuch mit Ihren Passwörtern verschlossen ist und nur von Ihnen gelesen werden kann.

Secrets im etcd

Auch wenn Secrets eine höhere Sicherheit bieten, sind sie standardmäßig unverschlüsselt im *etcd* zu finden. Das bedeutet, jeder mit API-Zugriff kann diese auch auslesen. Auch jeder Cluster-Admin hat Zugriff auf Ihre Secrets. Es ist also wichtig, dass Sie sich im Vorfeld Gedanken zu den Secrets machen, die Sie dort ablegen, und wie Sie den Zugang dazu einschränken wollen. Sie haben natürlich Optionen, um die Sicherheit zu erhöhen:

- Encryption at Rest einstellen
- RBAC-Zugriffsregeln implementieren
- Secret Access auf bestimmte Container einschränken

Bitte beachten Sie diese Punkte beim Einsatz von Kubernetes-Secrets, und besprechen Sie es am besten mit Ihren Cluster-Admins. Eine Anleitung zum Aktivieren von Encryption at Rest finden Sie hier:

https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/

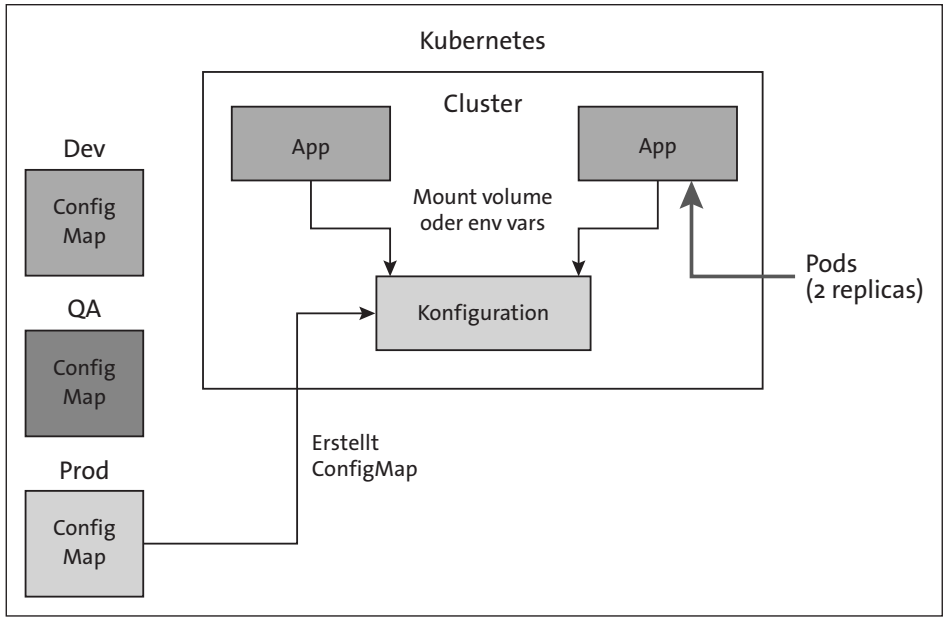

Abbildung 3.26 Konfigurationen je Umgebung

3.4.1 Was sind ConfigMaps?

Mithilfe der ConfigMap können Sie die Konfiguration Ihrer Anwendungen mit Kubernetes umsetzen. Dadurch entkoppeln Sie die Konfiguration und das Container-Image. Eine ConfigMap speichert in erster Linie Daten als Key-Value-Pairs, und Sie können diese dann in Ihren Container injizieren und in Ihrer Anwendung nutzen. Neben den klassischen Key-Value-Pairs können Sie in einer ConfigMap auch eine ganze Datei, wie eine Konfiguration in JSON, ablegen. Das ist für größere Konfigurationen sehr praktisch und lässt sich auch sehr gut in eine Anwendung einbinden.

[!]

Secrets statt ConfigMaps

ConfigMaps bieten gar keinen Schutz für sensible Daten. Nutzen Sie dafür lieber das Secrets-Objekt von Kubernetes.

Eine ConfigMap hat einen sehr einfachen Aufbau. In Listing 3.29 sehen Sie, dass die relevanten Felder für Ihre Daten `data` und `binaryData` sind. Dabei werden unter `data` normale Key-Value-Pairs abgelegt, und `binaryData` wurde für Base64 Encoded Strings entwickelt.

Doch um wirklich zu verstehen, was eine ConfigMap alles kann, werden wir gleich verschiedene Wege ausprobieren. Insgesamt gibt es vier Wege, um Daten aus einer

ConfigMap sinnvoll für Ihre Anwendung bereitzustellen. Der Inhalt von ConfigMaps kann

- von Pods als Dateisystem gemountet werden, damit die Anwendung die Datei auslesen kann.
- innerhalb des Pod-Manifests als Umgebungsparameter für die Anwendung gesetzt werden.
- für den Container als Befehlszeilenargument übergeben werden.
- innerhalb Ihrer Anwendung mithilfe der Kubernetes-API ausgelesen werden.

Es gibt wie immer nicht den perfekten Weg, um ConfigMaps einzubinden. Je nach Anwendungsfall und Art der Daten, die Sie in Ihrer Anwendung brauchen, ergibt der eine oder andere Ansatz mehr Sinn. In den nächsten Abschnitten werde ich Ihnen jede der vier Vorgehensweisen zeigen, damit Sie danach für Ihre Anwendung den richtigen Weg auswählen können.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-configmap
data:
  simpleKey: simpleValue
binaryData:
  binaryKey: dGVzdCBiaW5hcnkgZGF0YQ==
```

Listing 3.29 Einfaches ConfigMap-Manifest

Gut zu wissen

Sie können nicht unendlich viele Daten über eine ConfigMap einspielen. Eine ConfigMap darf die maximale Größe von 1 MiB nicht überschreiten.

ConfigMaps als Volume einbinden

Beginnen wir mit einer ConfigMap, die Sie als Volume einbinden werden. Dazu legen Sie die ConfigMap aus Listing 3.30 an. In dieser haben wir unter dem Key `config.json` ein vollständiges JSON-Objekt definiert. Wenn Sie diese ConfigMap ausrollen und sich in Lens anzeigen lassen, sollte es so aussehen wie in Abbildung 3.27. Sie sehen, dass das JSON-Objekt korrekt eingelesen wurde.

Das Thema Volumes werden Sie in Kapitel 6 noch deutlich detaillierter kennenlernen, doch ich erkläre Ihnen schon hier, was Sie für die ConfigMaps wissen müssen. Sie können für das Beispiel das Pod-Manifest aus Listing 3.31 nutzen.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-configmap
data:
  config.json: |
    {
        "key1": "value1",
        "key2": "value2"
    }
```

Listing 3.30 ConfigMap-Manifest mit einer Datei

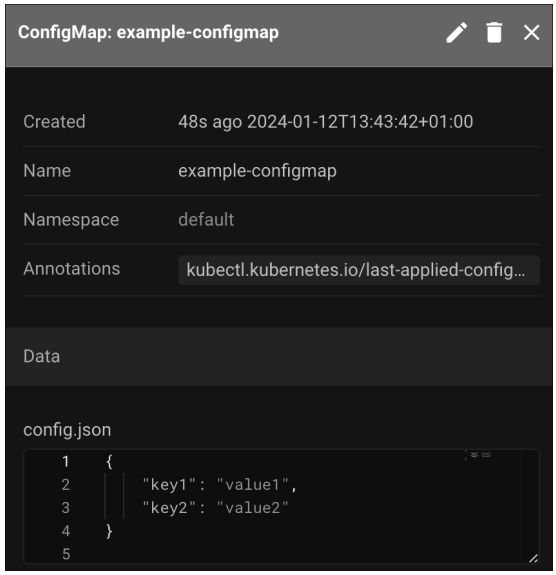

Abbildung 3.27 Eine ConfigMap in Lens anzeigen

Hier definieren wir unter `volumes` ein Volume mit dem Namen `config-volume` und verlinken auf die ConfigMap aus Listing 3.30 und das Item `config.json`. Dieses Volume wird unter `volumeMounts` unter dem `mountPath: /etc/config` gemountet, und Sie sollten dann die `config.json` im Container unter diesem Pfad finden.

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
    - name: example-container
```

```
    image: nginx
    volumeMounts:
    - name: config-volume
      mountPath: /etc/config
volumes:
  - name: config-volume
    configMap:
      name: example-configmap
      items:
      - key: "config.json"
        path: "config.json"
```

Listing 3.31 Pod-Manifest mit ConfigMap-Volume

In Abbildung 3.28 sehen Sie, wie das Volume vom Kubelet anhand der ConfigMap erzeugt wird. Auf dem Volume liegen die Daten, die Sie in der ConfigMap definiert haben.

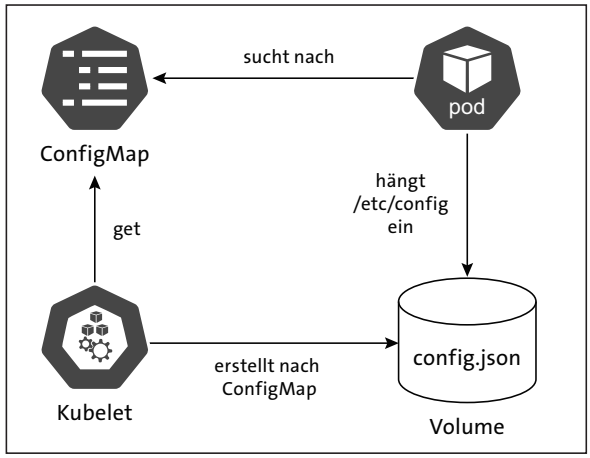

Abbildung 3.28 ConfigMap wird als Volume bereitgestellt

Rollen Sie das Pod-Manifest aus, und klinken Sie sich per `kubectl exec` auf den Pod. Jetzt können Sie den Inhalt des Volumes prüfen. Das Schöne am Einbinden als Volume ist, dass für jeden Key in der ConfigMap eine eigene Datei abgelegt wird, die den Value enthält. Das bedeutet, unter dem Pfad `/etc/config/config.json` finden Sie die Datei. Mit einem `cat` können Sie auch nachprüfen, ob der Inhalt stimmt.

Hinweis

Wie Sie in Listing 3.30 sehen, können Sie die Values auch mehrzeilig gestalten. Sie können die ganze Bandbreite von YAML verwenden. In Abschnitt 4.2 werden Sie mehr über YAML erfahren.

Lassen Sie uns als Nächstes die ConfigMap aus Listing 3.30 um ein weiteres Key-Value-Pair erweitern. Ich habe `test: "next one"` darunter eingefügt und das Update in Lens ausgerollt. Es wird jetzt eine kurze Zeit dauern, doch dann werden Sie im Volume Mount eine neue Datei mit dem Namen `test` und dem Inhalt `next one` finden.

Gut zu wissen

Wenn Sie eine ConfigMap updaten, dann werden die Werte in einem Volume Mount auch aktualisiert. Das Kubelet prüft periodisch, ob sich etwas verändert hat, und wird dann das Update auch in den Volume Mount einspielen. Es kann also einige Sekunden dauern, bis das Update auch im Pod ankommt.

Beim Einbinden von ConfigMaps als Umgebungsparameter werden Sie kein Update bekommen. In diesem Fall müssen Sie den Pod neu starten.

Das Kubelet wird in der aktuellen Pod-Definition jeden neuen Parameter der ConfigMap auch in das Volume einhängen, sobald es ein Update erkennt. Sie können jedoch im Pod-Manifest schon einschränken, welche Items Sie für Ihre Anwendung bereitstellen möchten. In Listing 3.32 finden Sie die entsprechende Erweiterung. Dadurch haben Sie deutlich mehr Kontrolle über den Mount, wenn Sie beispielsweise eine ConfigMap zwischen mehreren Anwendungen teilen.

```
volumes:
  - name: config-volume
    configMap:
      name: example-configmap
      items:
      - key: "config.json"
        path: "config.json"
```

Listing 3.32 Spezielle Auswahl an Keys einer ConfigMap

Hinweis

Wenn Sie ein Update der ConfigMap verhindern möchten, können Sie das mit einem Immutable-Tag wie in folgendem Beispiel erreichen.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-configmap
immutable: true
data:
  config.json: |
```

```
    {
        "key1": "value1",
        "key2": "value2"
    }
  test: "next one"
```

ConfigMaps als Umgebungsparameter einbinden

Eine weitere gängige Methode, um Konfigurationen an einer Anwendung durchzuführen, ist das Setzen von Umgebungsparametern. Auch die Werte einer ConfigMap können Sie nutzen, um Umgebungsparameter zu setzen. Ich würde in diesem Fall kein ganzes JSON-Objekt als Parameter übergeben, doch klassische Konfigurationen wie Log Level oder der Host-Name einer Datenbank sind hier genau richtig.

Für das Beispiel habe ich für Sie die ConfigMap aus Listing 3.33 und das Pod-Manifest aus Listing 3.34 vorbereitet. In dem Pod-Manifest wird durch `envFrom` eine Referenz auf die ConfigMap übergeben. Kubernetes wird dadurch alle Key-Value-Pairs als Umgebungsparameter setzen.

Probieren Sie es aus, und loggen Sie sich mit `kubectl exec` auf dem Pod ein. Mit dem Befehl `env` können Sie sich alle Umgebungsparameter ansehen und finden dort auch `LOG_LEVEL` und `DB_HOST`. Die Umgebungsparameter können Sie jetzt auch durch Ihre Anwendung einlesen lassen.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-env-configmap
data:
  LOG_LEVEL: "debug"
  DB_HOST: "localhost"
```

Listing 3.33 ConfigMap-Beispiel für Umgebungsparameter

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod-env
spec:
  containers:
    - name: example-container
      image: nginx
```

```
    envFrom:
    - configMapRef:
        name: example-env-configmap
```

Listing 3.34 Pod nutzt ConfigMap als Umgebungsparameter

Das ist ein gutes Beispiel, wenn Sie eine ConfigMap für eine Anwendung haben. Wie auch beim Einbinden als Volume können Sie die Auswahl der Umgebungsparameter noch genauer spezifizieren. Beispielsweise können Sie das Log Level aus einer anderen ConfigMap laden als den Datenbank-Host. Das ermöglicht Ihnen eine freiere Gestaltung Ihrer ConfigMaps.

In Listing 3.35 habe ich aus einer ConfigMap zwei gemacht, und in Listing 3.36 sehen Sie das Update des Pod-Manifests. Sie referenzieren in diesem Beispiel mit `valueFrom` einen expliziten Wert, der aus einer bestimmten ConfigMap gezogen werden soll, und da Sie den Wert von `PORT` aus der Datenbank-ConfigMap in Listing 3.35 nicht brauchen, müssen Sie diesen auch nicht unnötigerweise mitschleppen.

Gut zu wissen

Sie können in der ConfigMap nur String-Werte definieren. Aus diesem Grund ist der Parameter `PORT` aus Listing 3.35 auch kein Integer.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-env-configmap-log
data:
  LOG_LEVEL: "debug"
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-env-configmap-db
data:
  DB_HOST: "localhost"
  PORT: "1234"
```

Listing 3.35 Umgebungsparameter-ConfigMaps aufgeteilt

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod-env
```

```
spec:
  containers:
    - name: example-container
      image: nginx
      env:
        - name: DB_HOST
          valueFrom:
            configMapKeyRef:
              name: example-env-configmap-db
              key: DB_HOST
        - name: LOG_LEVEL
          valueFrom:
            configMapKeyRef:
              name: example-env-configmap-log
              key: LOG_LEVEL
```

Listing 3.36 Pod-Manifest mit ausgewählten Parametern aus verschiedenen ConfigMaps

Hinweis

Sie können eine ConfigMap auch immer als optional einbinden. Wenn die ConfigMap oder der Parameter nicht existieren, dann bleibt das eingebundene Volume oder die Umgebungsparameter leer.

```
- name: LOG_LEVEL
  valueFrom:
    configMapKeyRef:
      name: example-env-configmap-log
      key: LOG_LEVEL
      optional: true
```

Listing 3.37 Beispiel aus Listing 3.36

```
volumes:
  - name: config-volume
    configMap:
      name: example-configmap
      optional: true
```

Listing 3.38 Beispiel aus Listing 3.31

Container-Kommando durch ConfigMap übergeben

Diese Möglichkeit zeige ich Ihnen wegen der Vollständigkeit. Es gibt vermutlich Anwendungsfälle, bei denen Sie einem Pod ein Kommando übergeben möchten, das

durch eine ConfigMap konfiguriert werden kann. Doch bisher ist mir diese Notwendigkeit in der Praxis noch nie über den Weg gelaufen.

Sie können durch das Setzen eines Umgebungsparameters wie im vorherigen Abschnitt auch das Kommando für den Container parametrisieren. Für das Beispiel nutzen wir die ConfigMap aus Listing 3.33 und übergeben in Listing 3.39 einem Busybox-Image als Kommando ein Echo auf dem Parameter. Kubernetes wird den Pod starten, das Echo ausführen, und der Pod wird sich danach beenden.

Mit dem Befehl `kubectl logs example-pod` können Sie sehen, dass `localhost` ausgegeben wird. Also genau der Inhalt, den Sie in der ConfigMap unter dem Parameter `DB_HOST` definiert haben.

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
    - name: example-container
      image: busybox
      command: ["/bin/sh", "-c", "echo $HOST"]
      env:
        - name: HOST
          valueFrom:
            configMapKeyRef:
              name: example-env-configmap
              key: DB_HOST
  restartPolicy: Never
```

Listing 3.39 Container-Kommando als ConfigMap-Parameter übergeben

ConfigMaps über Kubernetes-API abfragen

Zu guter Letzt möchte ich Ihnen noch an einem kleinen Beispiel zeigen, wie Sie eine ConfigMap über die Kubernetes-API abfragen können. Das könnten Sie direkt in Ihrer Anwendung machen oder als Sidecar implementieren.

Hinweis

Wenn Sie die Kubernetes-API direkt aus der Anwendung nutzen wollen, dann müssen Sie sich in einem produktiven Umfeld auch über Berechtigungen und Zugangsdaten Gedanken machen. Die Anwendung benötigt dann einen technischen User, der die Berechtigung hat, die ConfigMap auszulesen.

Ich habe Ihnen dazu in Listing 3.40 ein Python-Script vorbereitet.

```
from kubernetes import client, config
config.load_kube_config()
v1 = client.CoreV1Api()
configmap_name = 'example-configmap'
namespace = 'default'
config_map = v1.read_namespaced_config_map(configmap_name, namespace)
print(config_map)
```

Listing 3.40 Python-Script zum Auslesen einer ConfigMap

Das Script nutzt das Kubernetes-Python-Paket, das Sie beispielsweise mit `pip install kubernetes` oder in Ihrer IDE downloaden können. Das Script macht Folgendes:

1. Es lädt Ihre Kubeconfig.
2. Es erstellt einen API-Client.
3. Es versucht, die ConfigMap aus Listing 3.30 aus dem `default`-Namespace zu laden.
4. Es gibt Ihnen die ConfigMap aus.

Als Ergebnis erhalten Sie die vollständige ConfigMap als JSON Payload. Mit dieser könnten Sie jetzt weiterarbeiten und sich die Daten daraus laden.

3.4.2 Was sind Secrets?

Kubernetes bietet für Secrets zusätzlichen Schutz im Vergleich zu ConfigMaps, da Secrets oft sensible Daten wie Passwörter, Tokens oder Zertifikate enthalten. Wenn wir in die Beispiele einsteigen, werden Sie jedoch sehen, dass die YAML-Syntax sehr ähnlich zu den ConfigMaps ist, was uns das Verwenden sehr leicht macht. Damit die Daten in Secrets gut geschützt sind, geht Kubernetes sehr sorgsam damit um. Beispielsweise wird das Secret

- nur an den Node gesendet, auf dem ein darauf angewiesener Pod läuft.
- vom Kubelet in einem temporären Dateisystem abgelegt, um zu verhindern, dass vertrauliche Daten dauerhaft gespeichert werden.
- vom Node gelöscht, sobald kein Pod darauf mehr das Secret benötigt.
- nur den Containern in einem Pod zugewiesen, denen Sie einen expliziten Zugriff gewähren.

Auch Sie können zur Sicherheit der Daten eines Secrets beitragen, indem Sie diese auch nach dem Auslesen in Ihrer Anwendung als Secret behandeln. Achten Sie auch beim Schreiben eines Manifests darauf, dass nur der Container Zugriff hat, der das Secret benötigt.

[+]

Gut zu wissen

Neben den Kubernetes-Secrets gibt es auch andere Produkte auf dem Markt, die Ihnen das Secret-Management erleichtern. Ich habe schon mit Kunden gearbeitet, die den AWS Secrets Manager oder HashiCorp Vault im Einsatz hatten. Jedes Produkt hat seine Berechtigung, und auch hier kann ich nur sagen: Es muss zu Ihrem Prozess passen.

Wenn Sie jedoch nur wegen des Injizierens von Passwörtern ein Vault Cluster einsetzen möchten, dann ist das Over Engineering. Sprechen Sie dazu am besten mit Ihren Cluster-Admins, um eine passende Lösung zu finden.

[!]

Vollzugriff durch priviligierte Container

Im normalen Fall kann ein Secret nur von einem Pod ausgelesen werden, dem Sie den Zugriff darauf explizit gewährt haben. Jedoch kann jeder Container, der mit der Option `privileged: true` gestartet wird, alle Secrets auslesen, die auf seinem Kubernetes-Node abgelegt sind.

Wenn Sie sich das Manifest aus Listing 3.41 ansehen, dann sehen Sie, dass der Aufbau der gleiche ist wie bei ConfigMaps.

```
apiVersion: v1
kind: Secret
metadata:
  name: example-secret
type: Opaque
data:
  username: YWRtaW4=
  password: cGFzc3dvcmQ=
```

Listing 3.41 Kubernetes-Secret-Manifest

Es gibt jedoch drei kleine Unterschiede:

1. Es gibt verschiedene Typen von Secrets, für die Sie eine Übersicht in Tabelle 3.8 finden.
2. Die Values unter `data` müssen immer Base64-encoded sein.
3. Wenn Sie Strings übergeben wollen, müssen Sie das unter `stringData` ablegen.

Generell können Sie wie ConfigMaps auch Secrets auf mehreren Wegen verwenden und in Ihre Pods injizieren. Sie können Secrets

- als Umgebungsparameter in einem Pod setzen.
- als Volume mounten.
- als Pull Secret für private Container Registries verwenden.

Gut zu wissen

Wie ConfigMaps können Sie auch Secrets

- optional einbinden.
- immutable anlegen.
- mit einer maximalen Größe von 1 MiB anlegen.

Typ	Beschreibung
Opaque	Standardtyp für Ihre Daten. Unterliegt keinen Vorgaben.
kubernetes.io/dockercfg	Enthält die serialisierte *~/.dockercfg*.
kubernetes.io/dockerconfigjson	Enthält die serialisierte *~/.docker/config.json*.
kubernetes.io/tls	Hier können Sie TLS-Zertifikate speichern. Ein gängiger Anwendungsfall ist, die Zertifikate für einen Ingress zu speichern.
kubernetes.io/ssh-auth	Enthält Zugangsdaten, die für eine SSH-Verbindung gebraucht werden.
bootstrap.kubernetes.io/token	Secrets, die beim Aufbau neuer Kubernetes-Nodes verwendet werden
kubernetes.io/basic-auth	Kubernetes prüft beim Erzeugen, ob die Keys `username` und `password` gesetzt sind, ansonsten gibt es keine weiteren Vorteile gegenüber dem Opaque-Typ. Es macht anderen Entwicklern direkt klar, wofür das Secret gedacht ist.
kubernetes.io/service-account-token	Hier werden Tokens von Service Accounts gespeichert. Dieser Token kann von einem Pod genutzt werden, um sich gegenüber der Kubernetes-API zu authentifizieren.

Tabelle 3.8 Secrets-Typen

[!]

Secrets im Repo

Secret-Manifeste sollten Sie nicht unverschlüsselt in einer Versionsverwaltung wie Git einchecken. Sie können Tools wie *sops* (*https://github.com/getsops/sops*) verwenden, um die Daten vorher zu verschlüsseln und beim Deployment zu entschlüsseln. Bitte klären Sie die Vorgehensweisen vorher mit Ihrem Unternehmen oder Ihren Cluster-Admins ab.

Secret als Umgebungsparameter einbinden

Das Setzen von Secrets als Umgebungsparameter funktioniert genauso wie bei ConfigMap. Lediglich die Syntax ist leicht anders, und die zeige ich Ihnen an einem kleinen Beispiel. Für unser Beispiel nutzen wir das Secret aus Listing 3.41 und das Pod-Manifest aus Listing 3.42. Die Veränderung zur ConfigMap habe ich Ihnen fett markiert.

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
  - name: example-container
    image: nginx
    env:
      - name: USERNAME
        valueFrom:
          secretKeyRef:
            name: example-secret
            key: username
      - name: PASSWORD
        valueFrom:
          secretKeyRef:
            name: example-secret
            key: password
```

Listing 3.42 Pod nutzt Secret als Umgebungsparameter

Wenn Sie die beiden Manifeste mit Lens ausrollen und sich den Pod ansehen, dann können Sie wie in Abbildung 3.29 in der Container-Übersicht auch die Environment-Parameter sehen, die gesetzt werden. Sie sehen auch, aus welchem Secret der Parameter kommt, und können sich sogar den Wert entschlüsselt anzeigen lassen.

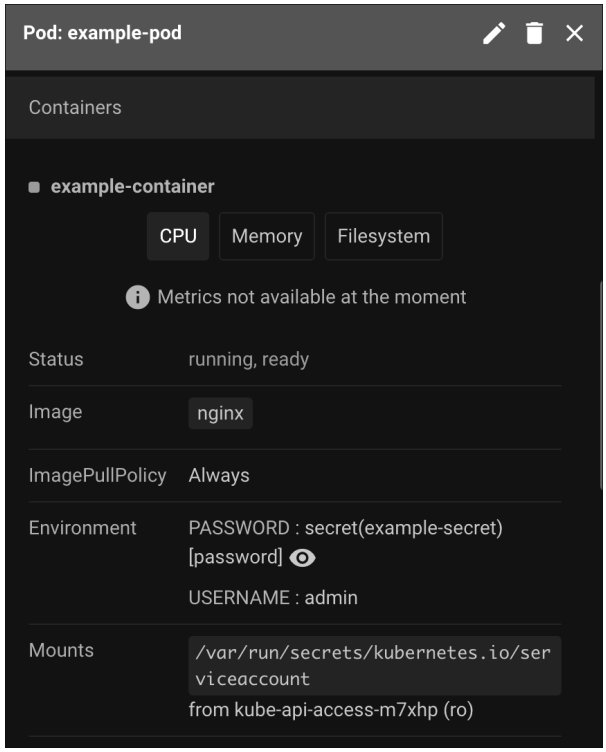

Abbildung 3.29 Secret-Parameter in der Pod-Übersicht

Gut zu wissen

Wie auch bei der ConfigMap können Sie alle Werte des Secrets als Umgebungsparameter setzen. Die Syntax sieht dann wie folgt aus:

```
envFrom:
  - secretRef:
    name: example-secret
```

Secret als Volume einbinden

Auch das Einbinden von Secrets als Volume funktioniert wie bei ConfigMaps. Eine sinnvolle Ergänzung ist es, wenn Sie die Secrets als Dotfile anlegen. Der Name der Datei beginnt mit einem `.` und wird so versteckt. In Listing 3.43 sehen Sie ein entsprechendes Secret-Manifest und in Listing 3.44 das passende Pod-Manifest.

Wenn Sie den Pod ausrollen und sich darauf mit `kubectl exec` einloggen, dann sollten Sie unter `/etc/secret-volume` eine Datei mit dem Namen `.secret-file` finden. Da es eine versteckte Datei ist, brauchen Sie den Befehl `ls -a`, um die Datei sehen zu können.

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  .secret-file: SGVsbG8gV29ybGQh
```

Listing 3.43 Dotfile-Secret

```
apiVersion: v1
kind: Pod
metadata:
  name: dotfile-test-pod
spec:
  containers:
    - name: dotfile-test-container
      image: nginx
      volumeMounts:
      - name: secret-volume
        mountPath: /etc/secret-volume
  volumes:
    - name: secret-volume
      secret:
        secretName: my-secret
```

Listing 3.44 Pod-Manifest mit Secret-Volume

Gut zu wissen

Auch beim Einbinden der Secrets als Volume können Sie spezielle Keys auswählen, die eingebunden werden sollen. Dazu passen Sie das Manifest einfach wie folgt an:

```
secret:
  secretName: my-secret
  items:
  - key: ".secret-file"
    path: ".secret-file"
```

Secrets anlegen mit kubectl

Wie die meisten anderen Kubernetes-Objekte auch, müssen Sie Secrets nicht unbedingt deklarativ anlegen. Sie können auch `kubectl` nutzen, um Secrets anzulegen. In diesem Fall kann es sogar sinnvoll sein, wenn Sie die Befehle innerhalb einer Pipeline

ausführen, weil Sie dort zur Laufzeit Passwörter entschlüsseln und in Kubernetes anlegen wollen. Aus diesem Grund stelle ich Ihnen hier diese Möglichkeit vor.

Hinweis

Ich würde ein verschlüsseltes YAML-File den imperativen Befehlen immer vorziehen, doch es gibt Situationen, in denen Sie die Befehle vielleicht gebrauchen können.

Die `kubectl`-Befehle variieren etwas, je nach Typ aus Tabelle 3.8, den Sie anlegen möchten. Doch die Grundstruktur bleibt gleich. Den Standardtyp Opaque können Sie beispielsweise wie in Listing 3.45 anlegen. Dort geben Sie einfach Key und Value direkt mit.

```
kubectl create secret generic my-secret \
    --from-literal=username=admin \
    --from-literal=password=secret
```

Listing 3.45 kubectl create secret from-literal

Sie können aber auch wie in Listing 3.46 auf Dateien verweisen, in denen der Value enthalten ist. Der Dateiname wird dann als Key verwendet.

```
kubectl create secret generic my-secret \
    --from-file=/path/to/username \
    --from-file=/path/to/password
```

Listing 3.46 kubectl create secret from-file

Secrets für private Container Registry

In Kubernetes können Sie Container nicht direkt von Ihrer Maschine aus deployen. Sie brauchen immer eine Container Registry, die Ihre Images verwaltet und die Kubernetes verwenden kann, um Images zu downloaden. Docker Hub ist eine öffentliche Registry, die wir in vielen Beispielen verwenden. Solange Ihr Kubernetes-Cluster netzwerktechnisch darauf zugreifen kann, können Sie auch Images von Docker Hub in Ihrem Cluster verwenden.

Wenn Sie in einem Unternehmen Software entwickeln und Container-Images ablegen wollen, dann werden Sie diese nicht öffentlich zugänglich machen wollen. Mit der Minikube Registry haben Sie schon einen ersten Eindruck gewonnen, wie eine private Registry funktioniert. Diese ist perfekt auf Minikube abgestimmt, und Sie brauchen sich in dem Kontext keine weiteren Gedanken zu machen. Der Betrieb einer privaten Registry in einem Unternehmen ist etwas herausfordernder, doch notwendig, denn in einer privaten Registry können Sie die Images eines Unterneh-

mens deutlich besser verwalten und haben mehr Kontrolle. Zudem bieten viele Registries noch zusätzliche Features wie Image Scans an. Produkte, mit denen ich in den letzten Jahren gearbeitet habe, sind:

- Artifactory von JFrog
- Nexus
- Amazon ECR

Doch es gibt viele weitere, die je nach Tech-Stack mehr oder weniger gut in ein Unternehmen passen. Das Wichtige bei einer Registry ist: Sie muss gut in den Entwicklungsprozess eingebunden sein, denn wenn sie kompliziert zu verwenden ist, werden doch wieder Passwörter in Textdateien gespeichert.

Gut zu wissen

Container Registries von Cloud-Anbietern wie Amazon ECR sind schnell und einfach innerhalb eines Accounts aufgebaut. Das führt in manchen Unternehmen dazu, dass die Container dezentral verwaltet werden, also jedes Team für sich die Container ablegt. Das ist in erster Linie weder gut noch schlecht, doch es sollte eine bewusste Entscheidung sein, ob die Container zentral oder dezentral gespeichert werden.

Ich persönlich finde die zentrale Speicherung für Produktions-Images besser, da Sie deutlich mehr Kontrolle darüber haben. So können Regeln wie

- Images dürfen nicht überschrieben werden
- Images dürfen nicht gelöscht werden
- Images müssen nach bekannten Sicherheitsproblemen gescannt werden

nicht einfach übergangen werden.

Private Registries sind in den meisten Fällen auch nicht ohne Authentifizierung zugänglich, und Kubernetes kann Images nicht einfach ziehen. Aus diesem Grund müssen Sie Kubernetes beibringen, sich zu authentifizieren. Mit Docker würden Sie für das Login einfach den Befehl `docker login` benutzen und Username und Passwort verwenden. Docker generiert daraus eine Konfigurationsdatei in JSON und speichert die Zugangsdaten dort nach dem Schema in Listing 3.47. Für jede Registry hält diese JSON einen String aus Username und Passwort, der Base64-encoded ist.

Gut zu wissen

Mit Docker Desktop sieht die Datei *config.json* etwas anders aus. Hier wird der `auth`-String nicht in der Datei abgelegt, und Sie finden nur den Hinweis in der JSON auf `"credsStore": "desktop"`.

```
{
    "auths": {
        "https://index.docker.io/v1/": {
            "auth": "g4s...3rda"
        }
    }
}
```

Listing 3.47 docker/config.json

Kubernetes basiert auf dieser Authentifizierung, und Sie können die Zugangsdaten aus der Konfigurationsdatei als Kubernetes-Secret ablegen und im Deployment-Manifest darauf referenzieren. Dafür können Sie entweder aus der `config.json` das Secret generieren, indem Sie folgenden Befehl verwenden:

```
kubectl create secret generic regcred \
    --from-file=.dockerconfigjson=<path/to/.docker/config.json> \
    --type=kubernetes.io/dockerconfigjson
```

Oder Sie erzeugen ein Secret und geben alle notwendigen Parameter im Befehl mit:

```
kubectl create secret docker-registry regcred \
    --docker-server=<url> \
    --docker-username=<username> \
    --docker-password=<password> \
    --docker-email=<emailadress>
```

Die zweite Variante ist aus meiner Sicht die bessere, denn Sie erzeugen das Secret explizit mit den Werten, die Sie brauchen. Zudem können Sie diesen Befehl auch in einer Deployment-Pipeline ausführen. Sie können auch ein Manifest wie in Listing 3.48 anlegen und die `config.json` dort Base64-encoded hinterlegen. Suchen Sie sich die Option aus, die für Sie am besten passt.

```
apiVersion: v1
data:
  .dockerconfigjson: eyJh…X0=
kind: Secret
metadata:
  name: regcred
type: kubernetes.io/dockerconfigjson
```

Listing 3.48 docker/config.json als Kubernetes-Secret

Hinweis

An dieser Stelle noch mal der Hinweis, dass Secrets nicht unverschlüsselt in der Versionsverwaltung abgelegt werden sollten. Sie können in einer CI/CD-Pipeline den Befehl zum Erstellen des Secrets ausführen und Passwörter zur Laufzeit einfügen, oder Sie nutzen ein zusätzliches Tool, um Secret-Manifeste zu verschlüsseln, bevor Sie diese in Git einchecken.

In einem Deployment-Manifest geben Sie das Secret als `imagePullSecret` mit. In Listing 3.49 sehen Sie die Option fett markiert. Den Namen setzen Sie auf den Namen, den Sie dem Secret mitgegeben haben, und schon kann Kubernetes mithilfe des Secrets auch Images aus einer privaten Registry ziehen.

Informationen im Secret

Außerhalb Ihres Testclusters sollten Sie in einem Kubernetes-Secret niemals Ihre privaten Zugangsdaten hinterlegen. Jeder, der Zugriff auf das Secret hat, kann auch Ihre Passwörter auslesen.

Nutzen Sie dafür einen technischen Benutzer, dessen Berechtigungen nur auf das eingeschränkt ist, was für den Anwendungsfall notwendig ist. Ein technischer Benutzer ist extra dafür da, einem System Zugang zu anderen Systemen zu geben.

Wie Sie einen erstellen können, unterscheidet sich von Registry zu Registry. Lesen Sie am besten in der entsprechenden Dokumentation, wie das Berechtigungskonzept funktioniert.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx-deployment
  labels:
    app: nginx
spec:
…
    spec:
      imagePullSecrets:
      - name: regcred
      containers:
      - name: my-container
        image: localhost:5000/my-nginx
        ports:
        - containerPort: 80
```

Listing 3.49 imagePullSecret in einem Deployment

Hinweis

Das Managen von Secrets ist nie leicht. Einer meiner Kunden betreibt seine Kubernetes-Cluster in AWS und nutzt dort den Secrets Manager, um Secrets abzulegen. Gleichzeitig nutzt er *sops* zum Verschlüsseln von Secrets in Gitlab, die er dann als Kubernetes-Secret anlegt.

Sobald es komplex und unübersichtlich wird, müssen Sie sich überlegen, wie es einfacher geht. Vor allem dann, wenn es keinen Single Point of Truth mehr gibt. Unter folgendem Link finden Sie einen Kubernetes-Operator, mit dem Sie externe Secret Stores ganz einfach anzapfen können. Das könnte Ihnen das Managen etwas vereinfachen.

https://github.com/external-secrets/external-secrets

3.5 Kommunikation mit Services und Ingress etablieren

Sie haben jetzt viel über einzelne Pods erfahren und wie Sie Ihre Anwendung in Kubernetes betreiben können. Doch ein Pod kommt selten allein, und in einer Welt von Microservices können das schnell mehrere Hundert werden. Die Herausforderung dabei ist, dass die Anwendungen untereinander kommunizieren wollen. Da die Pods in Kubernetes sehr volatil sind und Sie nicht wissen, auf welchem Node der Pod gerade läuft, brauchen Sie eine funktionierende Service Discovery.

Kubernetes bringt dafür das Service-Objekt mit. Dieses hält die Informationen über Ihre Pods und dient als Loadbalancer. Für die Kommunikation von außen in das Cluster hinein gibt es das Ingress-Objekt, das Ihnen ermöglicht, den Datenverkehr zu steuern. Beide Objekte sind wichtig und arbeiten zusammen, denn was wäre Ihre Anwendung, wenn niemand sie findet oder keiner sie erreichen kann?

Stellen Sie sich Ihre Anwendung vor wie ein Geschäft in einer Stadt. Sie haben sogar mehrere Läden mit dem gleichen Sortiment an mehreren Standorten. Das schätzen Ihre Kunden, denn gerade an einem Samstagnachmittag sind Ihre Läden brechend voll, und die Kunden können sich auf die Läden verteilen. Der Service ist Ihr schlaues Kundenleitsystem. Er kennt all Ihre Läden und führt genau Buch darüber, wo diese zu finden sind. Öffnet ein neuer Laden, dann leitet er Ihre Kunden auch an den Laden weiter.

Wenn der Service ein Kundenleitsystem ist, dann können Sie sich den Ingress als schlaues Parkleitsystem vorstellen, das Ihre Kunden von außerhalb der Stadt zum richtigen Parkplatz führt, wo dann das Kundenleitsystem (Service) übernimmt und den Kunden zu Ihrem Laden führt. Dabei fragt das Parkleitsystem am Anfang nach dem Ziel des Kunden und kann dabei sogar den Passierschein A38 abfragen und prüfen, ob er überhaupt berechtigt ist, in die Stadt zu fahren.

Hinweis

Genau für solche Fälle ist der Einsatz von Helm Diff perfekt. Sie ändern einen Wert, und in mehreren Dateien verändern sich dadurch die Templates. Ohne diese Tools wird so ein Zusammenspiel leicht übersehen.

Sie können mit Helm Diff nicht nur vor einem Release die Änderungen kontrollieren, sondern Sie können auch zwei Revisionen eines Releases miteinander vergleichen. Dadurch können Sie beispielsweise beim Debugging prüfen, was sich zum letzten Release verändert hat. Wenn Sie das Update mit dem Autoscaling eingespielt haben und vom Release des `humanity-backend` eine zweite Revision verfügbar ist, dann können Sie diesen Befehl verwenden:

```
helm diff revision humanity-backend 1 2
```

Dieses Kommando vergleicht Revision 1 mit Revision 2.

9.3 Eigene Charts entwickeln

Sie haben sich nun schon etwas mit Helm auseinandergesetzt und erste Helm-Charts in Kubernetes deployt. Jetzt wollen wir uns ansehen, wie Sie ein Helm-Chart für Ihre eigene Anwendung entwickeln und worauf Sie achten sollten.

Wie Sie wissen, können Sie ein Helm Chart auch ohne Repository verwenden und einfach im Git-Repo einchecken und in Kubernetes ausrollen. Dadurch können Sie von der Parametrisierung Gebrauch machen, doch Sie verlieren einen großen Vorteil: Die Wiederverwendbarkeit Ihres Charts in Ihrem Unternehmen.

Der eigentliche Entwicklungsprozess der Helm-Charts sieht aus wie in Abbildung 9.9.

Nehmen wir an, Sie entwickeln ein Helm-Chart für eine Postgres-Datenbank. Diese wird nicht nur von Ihnen verwendet, sondern könnte auch in anderen Unternehmensprojekten eingesetzt werden.

Anstatt sich in jedem Projekt eigenständig Gedanken zu einem perfekten Postgres-Setup zu machen, könnten Sie sich diese Gedanken einmal machen und für alle anderen ein Helm-Chart bereitstellen. Dieses Chart könnte auch als Inner-Sourcing-Projekt vorangetrieben werden, bei dem sich mehrere Teams Ihres Unternehmens beteiligen können.

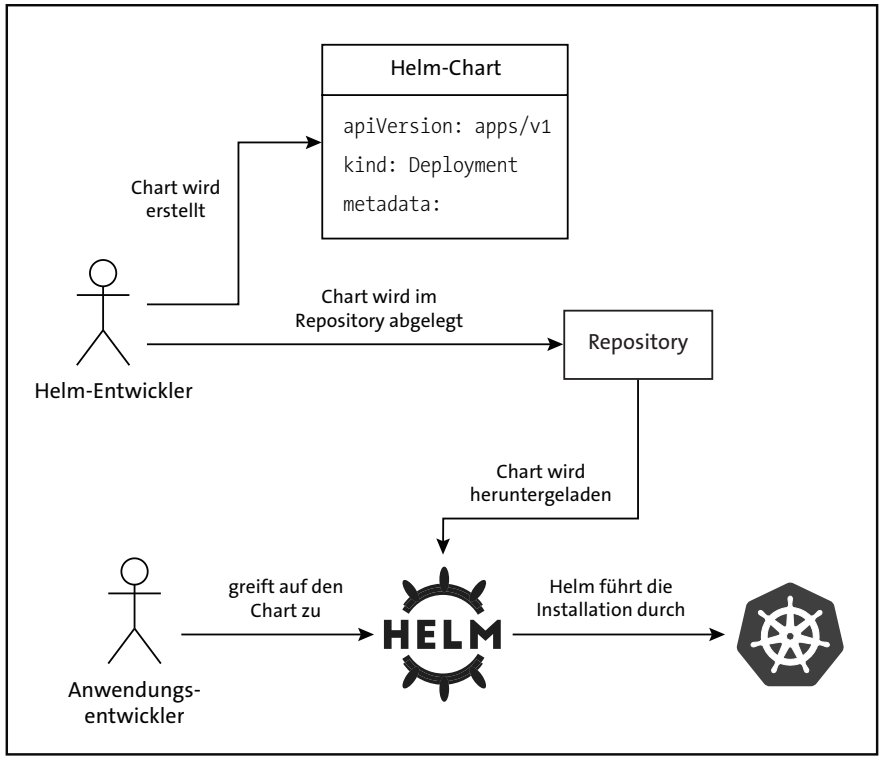

Abbildung 9.9 Entwicklungsprozess von Helm-Charts

Wenn Sie das Chart in einem Repository ablegen, können andere Entwickler es einfach nutzen und durch Values individualisieren. In diesem Kapitel sehen wir uns den Entwicklungsprozess genauer an. Also was Sie dafür brauchen, um ein Helm-Chart zu entwickeln, das von anderen verwendet werden kann. Zum Schluss gehen wir noch mal auf die Modularität von Helm ein und wie Sie durch Dependencies noch bessere Charts schreiben können.

9.3.1 Der Rahmen Ihres Helm-Charts

Damit Sie ein Chart entwickeln, das auch andere Entwickler nutzen wollen, brauchen Sie einen guten Rahmen. Zwei Dinge sind dafür wichtig:

- eine gut gepflegte Datei `Chart.yaml`
- hilfreiche Release Notes

Die Datei `Chart.yaml` enthält alle wesentlichen Metadaten über das Chart. Wenn Sie selbst ein Chart entwickeln, dann sollten Sie dieses auch pflegen. Sinnvolle Werte, die Sie ausfüllen sollten:

- `apiVersion`: Die API-Version des Charts, die Helm verwendet, um das Format und die Funktionalität des Charts zu interpretieren. In unseren Beispielen ist es v2 für Helm-3-Charts.
- `name`: Der Name des Helm-Charts. Dieser Name muss innerhalb eines Helm-Repositorys einzigartig sein.
- `version`: Die Version des Charts, die dem semantischen Versionierungsschema (*SemVer*) folgen muss.
- `description`: Eine kurze Beschreibung Ihres Charts und seiner Funktion.
- `keywords`: Eine Liste von Schlüsselwörtern, die mit dem Chart verbunden sind. Dies kann helfen, das Chart in Suchen zu finden.
- `home`: Eine URL zur Homepage des Projekts.
- `sources`: Eine Liste von URLs, die auf den Quellcode des im Chart verpackten Softwareprojekts verweisen.
- `dependencies`: Eine Liste der Abhängigkeiten von anderen Charts. Hier können Sie festlegen, welche anderen Charts benötigt werden, damit dieses Chart funktioniert. Die Abhängigkeiten sehen wir uns in Abschnitt 9.3.3 genauer an.
- `maintainers`: Eine Liste von zuständigen Entwicklern, damit dem Nutzer des Charts klar ist, wen er kontaktieren kann.

Hinweis

Es gibt noch weitere Optionen für die `Chart.yaml`. Die vollständige Übersicht finden Sie unter:

https://helm.sh/docs/topics/charts/

Die Release Notes haben Sie bei anderen Charts wie Jenkins schon kennengelernt. Dort sollten Sie Informationen reinpacken wie:

- **Hinweise zur Benutzerführung:** Klare Anweisungen, wie der Nutzer mit der deployten Anwendung interagieren kann.
- **Post-Deployment-Schritte:** Informieren Sie Ihre User über notwendige Schritte nach der Installation.
- **Wichtige Hinweise:** Teilen Sie Informationen mit, die für die Sicherheit, Konfiguration oder Nutzung des Charts relevant sind.

Im Falle von Jenkins war dies das Ausgeben des Standardpassworts, in Ihrem Fall könnte das vielleicht etwas anderes sein.

Release Notes werden in einer speziellen Datei namens `NOTES.txt` innerhalb des *templates*-Verzeichnisses Ihres Helm-Charts definiert. Die Syntax unterstützt die Tem-

plating-Engine von Helm, sodass Sie dynamisch Informationen basierend auf den Werten der Installation einfügen können.

Laden Sie sich das Jenkins-Chart einfach mal herunter, und öffnen Sie die Release Notes, um sich inspirieren zu lassen. In Listing 9.33 sehen Sie einen Ausschnitt daraus.

```
CHART NAME: {{ .Chart.Name }}
CHART VERSION: {{ .Chart.Version }}
APP VERSION: {{ .Chart.AppVersion }}
** Please be patient while the chart is being deployed **
{{- if .Values.ingress.enabled }}
1. Get the Jenkins URL and associate its hostname to your cluster external IP:
```

Listing 9.33 Ausschnitt aus der Datei NOTES.txt von Bitnami Jenkins

9.3.2 Charts verpacken und in Repository ablegen

Nachdem Sie Ihr eigenes Helm-Chart entwickelt und getestet haben, ist der nächste Schritt, dieses für die Verteilung und Nutzung vorzubereiten. Ein wesentlicher Bestandteil dieses Prozesses ist das Verpacken Ihres Charts und das Ablegen in einem Helm-Repository. Dadurch ist Ihr Chart leicht auffindbar, versioniert abgelegt und für andere Nutzer oder Teams innerhalb Ihrer Organisation zugänglich. Für unser Beispiel nutzen wir ChartMuseum, ein leichtgewichtiges, einfach zu bedienendes Tool, das als Repository für die Helm-Charts dient.

Ein Helm-Repository ist sehr einfach strukturiert. Sie können es sich als eine Sammlung von Paketen vorstellen, die Helm-Charts enthalten. Diese Repositories ermöglichen es Ihnen, Charts zu teilen und zu veröffentlichen, ähnlich wie Code in einem Git-Repository. Durch das Ablegen Ihres Charts in einem Repository sorgen Sie für

- eine einfache Versionierung: Jede Version Ihres Charts kann im Repository gespeichert werden, und Nutzer können dadurch auf spezifische Versionen zugreifen.
- eine einfache Verteilung: Entwicklerteams aus Ihrem Unternehmen können Ihre Charts einfach finden und verwenden.
- ein übersichtliches Abhängigkeitsmanagement: Charts können aufeinander aufbauen. Helm zieht die Abhängigkeiten aus den Repositories und kümmert sich um die einzelnen Schritte.

Installation ChartMuseum

Beginnen wir mit der Installation des ChartMuseums auf Minikube. Zum Installieren müssen Sie zuerst das Helm-Repository des ChartMuseums hinzufügen:

```
helm repo add chartmuseum https://chartmuseum.github.io/charts
helm repo update
```

Danach können wir das Helm-Chart über Lens installieren. Gehen Sie dafür wie in Abschnitt 9.1.4 vor. Sie sollten das Chart wie in Abbildung 9.10 dargestellt finden und installieren können. Es ist unbedingt notwendig, dass Sie in den Values des Chart-Museums den Parameter `DISABLE_API` auf `false` setzen. Nur so können Sie später über die API auch Helm-Charts hochladen.

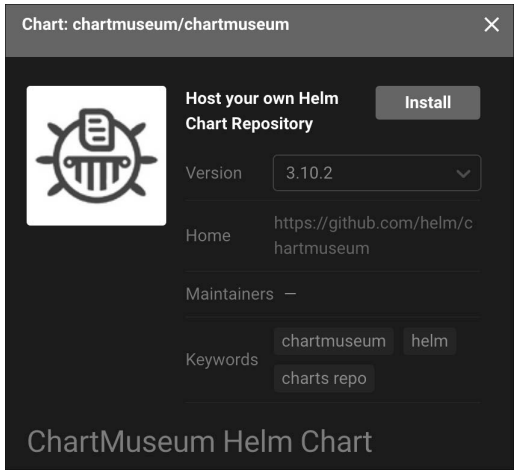

Abbildung 9.10 ChartMuseum über Lens installieren

Hinweis

Nach dem Hinzufügen mit `helm repo` müssen Sie gegebenenfalls Lens neu starten, damit es auch die neuen Helm-Repos scannt und das ChartMuseum anzeigt.

Nach der Installation leiten Sie einen Port über Lens auf den Service des ChartMuseums weiter. Ich habe dazu den Port 8080 von meiner Maschine weitergeleitet. Jetzt sind Sie bereit, Helm-Charts im ChartMuseum abzulegen.

Helm-Chart packen und hochladen

In diesem Beispiel nutze ich das `humanity-backend`, um es gleich zu packen und in das ChartMuseum zu laden. Gehen Sie dazu in den Ordner des Charts, und führen Sie folgende zwei Befehle aus:

```
helm package .
curl --data-binary "@humanity-backend-0.1.0.tgz" \
    http://localhost:8080/api/charts
```

Mit dem ersten Befehl wird aus Ihrem Helm-Chart das Archiv `humanity-backend-0.1.0.tgz` gemacht. Mit dem `curl`-Befehl laden Sie das Chart in das ChartMuseum hoch.

Hinweis

Helm wird beim Ausführen von `helm package` alle Dateien mitnehmen, die in dem Ordner liegen. Ähnlich wie bei Git berücksichtigt Helm jedoch die Datei `.helmignore`. Dort können Sie alle Ordner und Dateien einfügen, die Helm beim Erstellen des Pakets ignorieren soll.

Chart aus dem ChartMuseum verwenden

Abschließend wollen wir prüfen, ob das Chart auch wirklich im ChartMuseum liegt und wir es verwenden können. Dazu fügen Sie zuerst Ihr ChartMuseum als Repository zu Helm hinzu. Nutzen Sie einfach folgenden Befehl:

```
helm repo add my-chartmuseum http://localhost:8080/
```

Jetzt sollten Sie in Lens wie in Abbildung 9.11 unter den Helm-Charts auch Ihr Chart finden. Damit haben Sie erfolgreich ein eigenes Helm-Chart in ein privates Repository abgelegt. Versuchen Sie es zu installieren.

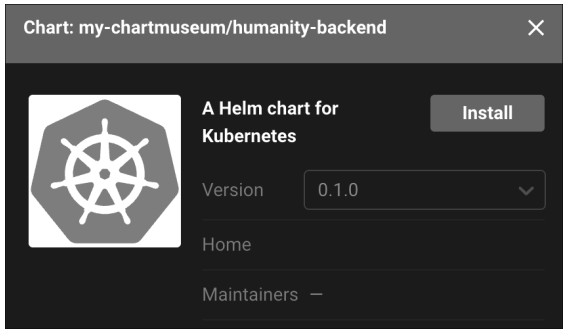

Abbildung 9.11 »humanity-backend«-Chart im ChartMuseum

Hinweis

Wir haben das ChartMuseum ohne Authentifizierung erstellt und genutzt. Sie sollten in Ihrem Unternehmen aber die Zugriffe darauf kontrollieren. Sprechen Sie dazu mit Ihren Cluster-Admins, denn vielleicht haben diese auch schon eine andere Repository-Lösung im Einsatz, die bereits an ein Identitätsmanagementsystem angeschlossen ist und damit eine Rechteverwaltung bietet. Denn es sollte offensichtlich sein: Wer auf die Helm-Charts zugreifen kann, kann weitreichende Manipulationen durchführen. Eine angemessene Absicherung ist abseits von Testsetups und Privatrechnern unbedingt notwendig.

9.3.3 Abhängigkeiten in Helm-Charts verwalten

Mit Helm haben Sie die Möglichkeit, Ihre Charts aufeinander aufbauen zu lassen. Durch eine definierte Abhängigkeit von einem anderen Chart können Sie die Helm-Charts modular aufbauen. Dieser Ansatz ermöglicht es, wiederverwendbare Komponenten zu schaffen, die in verschiedenen Projekten oder unter verschiedenen Bedingungen eingesetzt werden können. Es fördert die Wiederverwendung von Code, reduziert Redundanzen und erleichtert die Wartung von komplexen Kubernetes-Anwendungen. In diesem Abschnitt betrachten wir, wie Helm das Management von Abhängigkeiten zwischen Charts handhabt.

In Abbildung 9.12 sehen Sie, wie eine solche Abhängigkeit aussehen kann. Sie haben ein Chart Ihrer Anwendung versioniert in einem Repository abgelegt. Von dort aus können andere Charts darauf referenzieren und Ihre Anwendung in Ihrem Chart nutzen. Anwendungsfälle dafür sind beispielsweise:

- **Multi-Chart-Projekte:** Ein großes Chart, mit dem Sie alle Komponenten einer Anwendung deployen. Backend, Frontend und Datenbank werden komplett in einem Chart verwaltet und organisiert.
- **Individualisierung:** Sie möchten ein Chart nach Ihren Bedürfnissen individualisieren und zusätzliche Kubernetes-Ressourcen hinzufügen.

Wir werden uns im Laufe des Abschnitts beide Anwendungsfälle genauer ansehen.

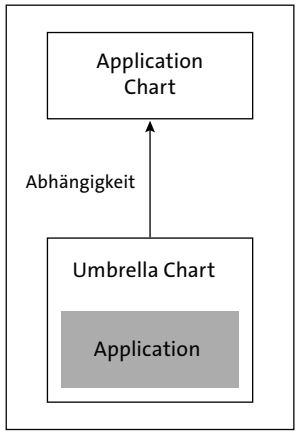

Abbildung 9.12 Abhängigkeiten von Charts

Gut zu wissen

Die Charts, die andere Abhängigkeiten nutzen, werden auch häufig *Umbrella Chart* genannt.

Hinzufügen und Aktualisieren von Abhängigkeiten

Als Beispiel für eine Abhängigkeit erweitern wir unser `humanity-backend` jetzt mit einer Postgres-Datenbank. Wir nutzen dazu das fertige Chart von Bitnami und tragen in die `Chart.yaml` die Abhängigkeit ein, wie Sie sie in Listing 9.34 finden.

```
dependencies:
  - name: postgresql
    version: "14.1.2"
    repository: "https://charts.bitnami.com/bitnami/"
```

Listing 9.34 Dependency in der »Chart.yaml«

Dann führen Sie folgenden Befehl aus:

```
helm dependency update
```

Helm wird jetzt mehrere Schritte durchführen. Zuerst wird das Helm-Chart der Postgres-Datenbank heruntergeladen und unter dem Pfad *charts* abgelegt. Danach wird Helm eine Datei mit dem Namen `Chart.lock` anlegen, die aussieht wie in Listing 9.35.

```
dependencies:
- name: postgresql
  repository: https://charts.bitnami.com/bitnami/
  version: 14.1.2
digest: sha256:9133c60dc762bdd233266d780db912857f04f6033503fc4032ac43be17d18f
generated: "2024-02-21T09:58:03.050996+01:00"
```

Listing 9.35 Chart.lock

Diese Datei speichert die installierte Abhängigkeit mit einem Hashwert, den Sie später beispielsweise in der Pipeline nutzen können, um die Abhängigkeiten mit der gleichen Version wieder zu installieren. Mit dem Befehl `helm dependency build` können Sie genau die Version installieren, die im `Chart.lock` festgehalten wurde. Dadurch müssen Sie nicht das Subchart ins Git-Repo einchecken, aber können sich sicher sein, dass nicht die falsche Version installiert wird.

Wenn Sie jetzt ein Update des Charts durchführen möchten, müssen Sie die Version der Postgres im `Chart.yaml` erhöhen. In meinem Fall erhöhe ich den Wert auf die Version 14.1.3 und führe den Befehl `helm dependency update` erneut aus. Das alte Helm-Chart wird mit der neuen Version ersetzt, und die `Chart.lock` wird aktualisiert.

Sie sollten die Abhängigkeiten regelmäßig aktualisieren, damit Sie auch Sicherheitsupdates bekommen. Leider ist jeder Update-Prozess eines Subcharts etwas aufwendig, denn Sie müssen sich die Änderungen ansehen und schauen, ob sich in der

Konfiguration etwas verändert hat. Das kann bei großen Versionssprüngen sehr aufwendig sein.

Gut zu wissen

Die URL des Repositorys ist die gleiche, die Sie auch mit `helm repo ls` finden, weil wir sie dort als lokales Repo eingetragen haben. Helm braucht hier die richtige URL zum Repository, da der Name des lokalen Repos von Entwickler zu Entwickler unterschiedlich sein kann.

Subcharts konfigurieren

Die Konfiguration der Subcharts erfolgt primär durch die `values.yaml` des Hauptcharts. Jedes Subchart bringt seine eigene `values.yaml` mit, die Standardwerte für das Subchart definiert. Um diese Standardwerte zu überschreiben oder anzupassen, definieren Sie im Hauptchart Werte für die Subcharts in Ihrer eigenen `values.yaml`. Dazu nutzen Sie als Schlüssel den Namen des Subcharts.

In unserem Beispiel mit der Postgres-Datenbank können Sie in der `values.yaml` die Konfiguration aus Listing 9.36 hinzufügen, um Username, Passwort und den Namen der Datenbank anzupassen. Der oberste `postgresql`-Key verweist dabei auf das Subchart, die darunterliegenden Schlüssel finden Sie so in der `values.yaml` des Postgres-Deployments.

```
postgresql:
  global:
    postgresql:
      auth:
        username: "kevinwelter"
        password: "test1234"
        database: "database"
```

Listing 9.36 Subchart-Konfiguration in »values.yaml«

Wenn Sie Ihr Chart jetzt ausrollen, können Sie wie in Abbildung 9.13 in dem Postgres-Pod sehen, dass die Parameter übernommen wurden.

Hinweis

Durch das Installieren der Abhängigkeit liegt unter *charts* das Subchart. Dort können Sie sich einfach die `values.yaml` ansehen und herausfinden, welche Konfigurationen Sie anpassen möchten.

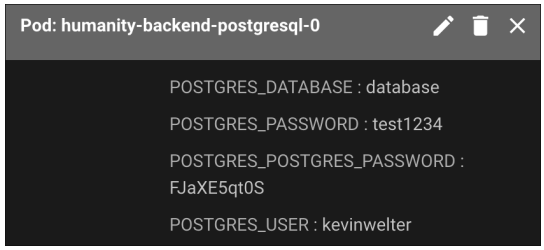

Abbildung 9.13 Umgebungsparameter Postgres

Multi-Chart-Projekt

Ein praktisches Beispiel für die Nutzung von Abhängigkeiten ist ein Helm-Chart, das als Wrapper für mehrere Subcharts dient. Dies ermöglicht es Ihnen, eine Anwendung, die aus mehreren unabhängigen Komponenten besteht, als ein Ganzes zu deployen und zu verwalten. Die Struktur könnte dann so aussehen wie in Listing 9.37. In diesem Beispiel besteht `my-application` aus zwei Subcharts, `frontend` und `backend`, die als Abhängigkeiten in der Datei `Chart.yaml` von `my-application` definiert sind. Benutzer können `my-application` installieren, und Helm kümmert sich um das Deployment der Subcharts `frontend` und `backend` basierend auf deren Konfigurationen.

```
my-application/
├── Chart.yaml
├── values.yaml
└── charts/
    ├── frontend/
    │   └── Chart.yaml
    │   ...
    └── backend/
        └── Chart.yaml
        ...
```

Listing 9.37 Beispielstruktur Multi-Chart-Projekt

Charts individualisieren

Neben der Nutzung von Subcharts und Multi-Chart-Projekten gibt es Situationen, in denen Sie ein Helm-Chart spezifisch nach Ihren Bedürfnissen individualisieren möchten. Dies kann der Fall sein, wenn Sie zusätzliche Kubernetes-Ressourcen integrieren oder bestehende Konfigurationen anpassen müssen, die nicht direkt durch das Standard-Chart abgedeckt werden.

Bei einem Kunden haben wir beispielsweise das Standard-Chart des Tools Kyverno mit Policies angereichert, die global für das Unternehmen geprüft werden sollen. Auf

diesen »Unternehmens-Kyverno« haben dann einzelne Projekte aufgesetzt und zusätzliche Policy-Erweiterungen implementiert, die dann in den Clustern ausgerollt wurden.

Ein weiteres Beispiel ist die Erweiterung einer Anwendung um ConfigMaps für das Monitoring. Dafür habe ich auf einem Standard-Chart als Dependency aufgesetzt und zusätzliche ConfigMaps für den Prometheus-Operator erzeugt. Dadurch konnte das Standard-Chart direkt mit Metriken und Alarmen ausgeliefert werden.

Listing 9.38 zeigt beispielhaft, wie so ein Setup aussehen kann.

```
my-kyverno/
├── Chart.yaml
├── values.yaml
├── charts/
│   └── kyverno/
│       ├── Chart.yaml
│       └── ... # Weitere Dateien und Ordner des Kyverno-Charts
└── templates/
    ├── policies.yaml # Ihre individuellen Policies
    └── prometheus-metrics.yaml  # Ihre ConfigMap für Prometheus
```

Listing 9.38 Ordnerstruktur für Chart-Individualisierung

9.4 Fazit

Jetzt kennen Sie sich mit Helm-Charts und deren Abhängigkeiten aus und haben die Werkzeuge in der Hand, um Ihre Kubernetes-Anwendungen effektiv zu managen. Die Möglichkeit, Abhängigkeiten in Ihren Helm-Charts zu definieren, hilft Ihnen, Ihre Anwendungen modular und wartbar zu gestalten. Sie müssen die Charts nicht neu erfinden, sondern können sich auf andere Projekte verlassen und durch Individualisierungen Ihre Konfiguration ergänzen und anpassen.

Mit den jetzt erlernten Fähigkeiten können Sie Ihre Kubernetes-Anwendungen mit einer neuen Perspektive angehen. Legen Sie dabei den Fokus auf

- Modularität,
- Wiederverwendbarkeit und
- einfache Verwaltung

und nutzen Sie diese Möglichkeiten, um Ihre Anwendungen besser zu strukturieren und zu verwalten. Mit Helm haben Sie die Kontrolle und die Flexibilität, um Ihre Deployments genau nach Ihren Vorstellungen zu gestalten. Happy Helming!