Kubernetes Best Practices

Praktische Anleitungen und Vorlagen zu Grundlagen und fortgeschrittenen Themen

1 Einen einfachen Service einrichten

Dieses Kapitel beschreibt das Verfahren zum Einrichten einer einfachen mehrschichtigen Anwendung in Kubernetes. Das Beispiel, das wir durchgehen werden, besteht aus zwei Schichten: einer einfachen Webanwendung und einer Datenbank. Auch wenn es sich hierbei nicht um die komplizierteste Anwendung handelt, ist es ein guter Ausgangspunkt, um die Verwaltung einer Anwendung in Kubernetes zu erlernen.

1.1 Die Anwendung im Überblick

Die Anwendung für unser Beispiel ist recht simpel. Es handelt sich um einen einfachen Journaldienst, der folgende Details aufweist:

- Die Anwendung hat einen separaten statischen Dateiserver, der NGINX verwendet.
- Sie verfügt über ein RESTful Application Programming Interface (API) *https://some- host-name.io/api* im Pfad */api*.
- Sie besitzt auf der Haupt-URL *https://some-host-name.io* einen Dateiserver.
- Sie verwendet den Dienst Let's Encrypt (*https://letsencrypt.org*) für die Verwaltung von Secure Sockets Layer (SSL).

Abbildung 1–1 zeigt ein Diagramm dieser Anwendung. Machen Sie sich keine Sorgen, wenn Sie nicht auf Anhieb alle einzelnen Teile verstehen; sie werden im Laufe des Kapitels detaillierter erklärt. Wir werden den Aufbau dieser Anwendung Schritt für Schritt durchgehen, und verwenden zunächst YAML-Konfigurationsdateien und dann Helm-Charts.

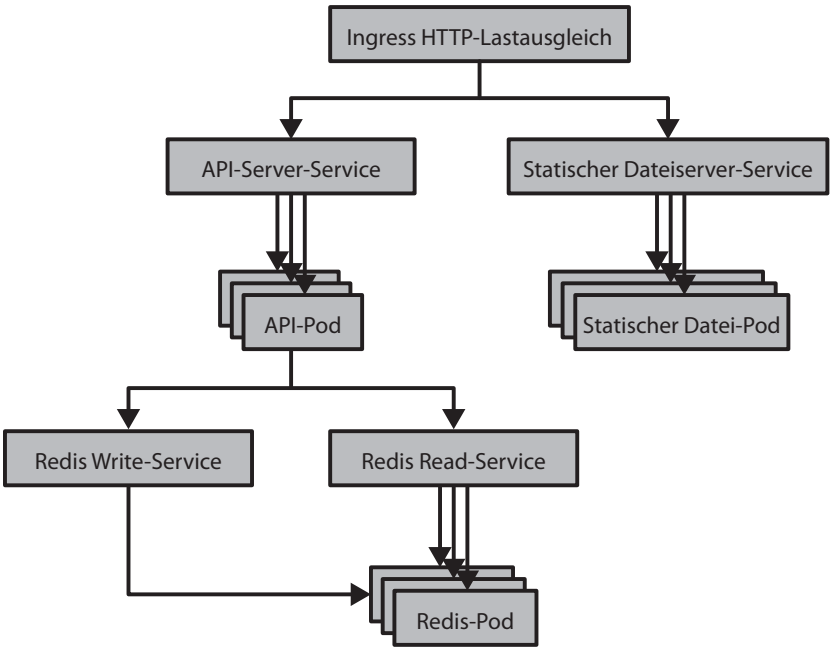

Abb. 1–1 *Ein Diagramm unseres Journaldienstes, wie er in Kubernetes implementiert ist*

1.2 Konfigurationsdateien verwalten

Bevor wir uns im Detail mit dem Aufbau dieser Anwendung in Kubernetes befassen, sollten wir beschreiben, wie wir die Konfigurationen selbst verwalten. Bei Kubernetes wird alles *deklarativ* repräsentiert. Das bedeutet, dass Sie den gewünschten Zustand der Anwendung im Cluster aufschreiben (im Allgemeinen in YAML- oder JSON-Dateien). Diese deklarierten gewünschten Zustände definieren alle Teile Ihrer Anwendung. Dieser deklarative Ansatz ist einem *imperativen* Ansatz vorzuziehen, bei dem der Zustand Ihres Clusters die Summe einer Reihe von Änderungen am Cluster ist. Wenn ein Cluster imperativ konfiguriert ist, ist es schwierig zu verstehen und zu reproduzieren, wie er in diesen Zustand gekommen ist. Hierdurch ist es dann auch schwierig, Probleme mit Ihrer Anwendung zu verstehen oder zu beheben.

Bei der Deklaration des Zustands Ihrer Anwendung wird in der Regel YAML gegenüber JSON bevorzugt, obwohl Kubernetes beide Formate unterstützt. Der Grund dafür ist, dass YAML etwas weniger ausführlich ist und von Menschen besser bearbeitet werden kann als JSON. Es ist jedoch zu beachten, dass YAML auf Einrückungen reagiert; häufig lassen sich Fehler in Kubernetes-Konfigurationen auf eine falsche Einrückung in YAML zurückführen. Wenn sich die Dinge nicht wie erwartet verhalten, ist die Überprüfung der Einrückungen ein guter Ansatzpunkt für die Fehlersuche. Die meisten Editoren unterstützen die Syntaxhervorhebung sowohl für JSON als auch für YAML. Wenn Sie mit diesen Dateien arbeiten, ist es eine gute Idee, solche

Tools zu installieren, um in Ihren Konfigurationen sowohl Autoren- als auch Dateifehler leichter zu finden. Es gibt auch eine hervorragende Erweiterung für Visual Studio Code, die eine umfassendere Fehlerprüfung für Kubernetes-Dateien unterstützt.

Da der deklarative Zustand, der in diesen YAML-Dateien enthalten ist, als Source of Truth für Ihre Anwendung dient, ist die korrekte Verwaltung dieses Zustands entscheidend für den Erfolg Ihrer Anwendung. Wenn Sie den gewünschten Zustand Ihrer Anwendung ändern, müssen Sie in der Lage sein, die Änderungen zu verwalten, zu überprüfen, ob sie korrekt sind, zu kontrollieren, wer die Änderungen vorgenommen hat, und im Falle eines Fehlers die Änderungen zurückzunehmen. Glücklicherweise haben wir im Rahmen der Softwareentwicklung bereits die notwendigen Werkzeuge entwickelt, um sowohl Änderungen am deklarativen Zustand als auch die Prüfung und das Rollback zu verwalten. Die bewährten Verfahren zur Versionsverwaltung und für Code-Reviews lassen sich direkt auf die Verwaltung des deklarativen Zustands Ihrer Anwendung anwenden.

Heutzutage speichern die meisten Leute ihre Kubernetes-Konfigurationen in Git. Auch wenn die spezifischen Details des Versionskontrollsystems unwichtig sind, erwarten viele Tools im Kubernetes-Ökosystem Dateien in einem Git-Repository. Beim Code-Review gibt es eine viel größere Heterogenität; obwohl GitHub offensichtlich sehr beliebt ist, verwenden andere On-Premises-Tools oder Dienste für das Code-Review. Unabhängig davon, wie Sie das Code-Review für die Konfiguration Ihrer Anwendung implementieren, sollten Sie sie mit der gleichen Sorgfalt und Konzentration behandeln, die Sie für die Verwaltung des Quellcodes anwenden.

Wenn es darum geht, das Dateisystem für Ihre Anwendung einzurichten, lohnt es sich, die mit dem Dateisystem gelieferte Verzeichnisstruktur zu verwenden, um Ihre Komponenten zu organisieren. In der Regel wird ein einzelnes Verzeichnis genutzt, um einen Anwendungsdienst zusammenzufassen. Die Definition dessen, was einen Anwendungsdienst ausmacht, kann von Team zu Team unterschiedlich sein, aber im Allgemeinen handelt es sich um einen Service, der von einem Team aus 8 bis 12 Personen entwickelt wird. Innerhalb dieses Verzeichnisses werden Unterverzeichnisse für Unterkomponenten der Anwendung eingesetzt.

Für unsere Anwendung legen wir die Verzeichnisstruktur für die Dateien wie folgt an:

```
journal/
  frontend/
  redis/
  fileserver/
```

In jedem Verzeichnis befinden sich die konkreten YAML-Dateien, die zur Definition des Service benötigt werden. Wie Sie später sehen werden, wird dieses Dateilayout komplizierter, wenn wir beginnen, unsere Anwendung in mehreren verschiedenen Regionen oder Clustern einzusetzen.

1.3 Mit Deployments einen replizierten Service erstellen

Um unsere Anwendung zu beschreiben, beginnen wir mit dem Frontend und arbeiten uns nach unten vor. Die Frontend-Anwendung für das Journal ist eine in TypeScript implementierte Node.js-Anwendung. Die komplette Anwendung ist zu umfangreich, um sie in das Buch aufzunehmen, daher haben wir sie auf unseren GitHub hochgeladen (*https://github.com/brendandburns/kbp-sample*). Dort finden Sie auch den Code für zukünftige Beispiele, es lohnt sich also, für diese URL ein Lesezeichen zu setzen. Die Anwendung stellt einen HTTP-Dienst auf Port 8080 bereit, der Anfragen an den Pfad */api/** weiterleitet und das Redis-Backend verwendet, um die aktuellen Journaleinträge hinzuzufügen, zu löschen oder zurückzugeben. Wenn Sie die folgenden YAML-Beispiele auf Ihrem lokalen Rechner durcharbeiten möchten, sollten Sie diese Anwendung mithilfe des Dockerfiles in ein Container-Image erstellen und in Ihr eigenes Image-Repository pushen. Anstatt den Namen unserer Beispieldatei zu nehmen, sollten Sie dann in Ihren Code den Namen Ihres Container-Image aufnehmen.

1.3.1 Best Practices für die Verwaltung von Images

Obwohl die Erstellung und Pflege von Container-Images im Allgemeinen den Rahmen dieses Buches sprengen würde, lohnt es sich, einige allgemeine Best Practices für die Erstellung und Benennung von Images aufzuzeigen. Im Allgemeinen kann der Image-Erstellungsprozess für »Supply-Chain-Angriffe« anfällig sein. Bei solchen Angriffen injiziert ein böswilliger Benutzer Code oder Binärdateien in eine Abhängigkeit von einer vertrauenswürdigen Quelle, die dann in Ihre Anwendung eingebaut wird. Aufgrund des Risikos solcher Angriffe ist es wichtig, dass Sie bei der Erstellung Ihrer Images nur auf bekannte und vertrauenswürdige Image-Anbieter zurückgreifen. Alternativ können Sie auch alle Ihre Images von Grund auf neu erstellen. Bei einigen Sprachen (z. B. Go), die statische Binärdateien erstellen können, ist die Erstellung von Grund auf einfach, bei interpretierten Sprachen wie Python, JavaScript oder Ruby ist dies jedoch wesentlich komplizierter.

Die anderen Best Practices für Images beziehen sich auf die Namensgebung. Obwohl die Version eines Container-Images in einer Image-Registry theoretisch veränderbar ist, sollten Sie das Versions-Tag als unveränderbar behandeln. Insbesondere ist eine Kombination aus der semantischen Version und dem SHA-Hash des Commits, in dem das Image erstellt wurde, eine gute Praxis für die Benennung von Images (z. B. *v1.0.1-bfeda01f*). Wenn Sie keine Image-Version angeben, wird standardmäßig `latest` verwendet. Obwohl dies in der Entwicklung praktisch sein kann, ist es eine schlechte Idee für den produktiven Einsatz, da `latest` jedes Mal, wenn ein neues Image erstellt wird, verändert wird.

1.3.2 Replizierte Anwendung erstellen

Unsere Frontend-Anwendung ist *zustandslos*; sie verlässt sich in Bezug auf ihren Zustand vollständig auf das Redis-Backend. Daher können wir sie beliebig replizieren, ohne den Datenverkehr zu beeinträchtigen. Obwohl es unwahrscheinlich ist, dass unsere Anwendung in großem Umfang genutzt wird, ist es dennoch eine gute Idee, mit mindestens zwei Replikaten zu arbeiten, damit Sie einen unerwarteten Absturz bewältigen oder eine neue Version der Anwendung ohne Ausfallzeiten ausrollen können.

In Kubernetes ist die ReplicaSet-Ressource diejenige, die direkt die Replikation einer bestimmten Version Ihrer containerisierten Anwendung verwaltet. Da sich die Version aller Anwendungen im Laufe der Zeit ändert, wenn Sie den Code ändern, ist es keine optimale Methode, direkt ein ReplicaSet zu verwenden. Stattdessen nutzen Sie die Ressource *Deployment*. Ein Deployment kombiniert die Replikationsfunktionen von ReplicaSet mit der Versionierung und der Möglichkeit, ein gestuftes Rollout durchzuführen. Durch ein Deployment können Sie die in Kubernetes integrierten Werkzeuge nutzen, um von einer Version der Anwendung zur nächsten zu wechseln.

Die Kubernetes-Deployment-Ressource für unsere Anwendung sieht wie folgt aus:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    # Alle Pods im Deployment verwenden dieses Label
    app: frontend
  name: frontend
  namespace: default
spec:
  # Wir sollten aus Gründen der Zuverlässigkeit immer mindestens zwei Replikate
einsetzen
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - image: my-repo/journal-server:v1-abcde
        imagePullPolicy: IfNotPresent
        name: frontend
        # TODO: Ermitteln, wie hoch der tatsächliche Ressourcenbedarf ist
        resources:
          request:
            cpu: "1.0"
            memory: "1G"
          limits:
            cpu: "1.0"
            memory: "1G"
```

Bei diesem Deployment sind mehrere Dinge zu beachten. Erstens verwenden wir Labels, um das Deployment sowie die ReplicaSets und die Pods zu identifizieren, die das Deployment erstellt. Wir haben das Label `app: frontend` zu all diesen Ressourcen hinzugefügt, damit wir alle Ressourcen für eine bestimmte Schicht in einer einzigen Anfrage untersuchen können. Sie werden sehen, dass wir beim Hinzufügen weiterer Ressourcen genauso vorgehen werden.

Außerdem haben wir an einigen Stellen in das YAML Kommentare eingefügt. Obwohl diese Kommentare nicht in die auf dem Server gespeicherte Kubernetes-Ressource einfließen, dienen sie, genau wie Kommentare im Code, als Orientierungshilfe für Personen, die sich diese Konfiguration zum ersten Mal ansehen.

Sie sollten weiterhin beachten, dass wir für die Container im Deployment für die Ressourcenanforderungen sowohl Request als auch Limit angegeben und für Request und Limit die gleichen Werte verwendet haben. Wenn eine Anwendung ausgeführt wird, ist Request (Anforderung) die Reservierung, die auf dem Host-Rechner, auf dem sie läuft, garantiert wird. Limit (Obergrenze) ist die maximale Ressourcennutzung, die dem Container zugestanden wird. Wenn Sie am Anfang Request gleich Limit setzen, führt dies zu einem möglichst vorhersehbaren Verhalten Ihrer Anwendung. Diese Vorhersagbarkeit geht jedoch zulasten der Ressourcennutzung. Da die Einstellung Request gleich Limit Ihre Anwendungen daran hindert, zu viel Zeit einzuplanen oder zu viele ungenutzte Ressourcen zu verbrauchen, werden Sie nicht in der Lage sein, eine maximale Auslastung zu erreichen, es sei denn, Sie stimmen Request und Limit sehr, sehr sorgfältig aufeinander ab. Wenn Sie das Kubernetes-Ressourcenmodell besser verstehen, können Sie Request und Limit für Ihre Anwendung unabhängig voneinander anpassen. Die meisten Benutzer vertreten jedoch die Auffassung, dass eine stabile Vorhersagbarkeit die geringere Ressourcenausnutzung wert ist.

Wie unser Kommentar andeutet, ist es oft schwierig, die richtigen Werte für diese Ressourcenobergrenzen zu ermitteln. Ein guter Ansatz ist, die Schätzungen zunächst zu hoch anzusetzen und sie dann mithilfe des Monitorings auf die richtigen Werte abzustimmen. Wenn Sie jedoch einen neuen Service einführen, sollten Sie bedenken, dass Ihr Ressourcenbedarf beim ersten großen Datenverkehr wahrscheinlich erheblich ansteigen wird. Darüber hinaus gibt es einige Sprachen, insbesondere Sprachen mit Garbage Collection, die munter den gesamten verfügbaren Speicher verbrauchen, was es schwierig machen kann, das richtige Minimum für den Speicher zu bestimmen. In diesem Fall kann eine Form der binären Suche notwendig sein, aber denken Sie daran, dies in einer Testumgebung zu tun, damit es sich nicht auf Ihre Produktivumgebung auswirkt!

Nachdem wir nun die Deployment-Ressource definiert haben, checken wir sie in die Versionskontrolle ein und stellen sie in Kubernetes bereit:

```
git add frontend/deployment.yaml
git commit -m "Deployment hinzugefügt" frontend/deployment.yaml
kubectl apply -f frontend/deployment.yaml
```

Es ist ebenfalls eine Best Practice, sicherzustellen, dass der Inhalt Ihres Clusters genau mit dem Inhalt Ihrer Quellcodeverwaltung übereinstimmt. Das beste Muster dafür ist die Verwendung eines GitOps-Ansatzes und die Bereitstellung für die Produktion nur von einem bestimmten Zweig Ihrer Versionsverwaltung aus vorzunehmen. Hierzu nutzen Sie die Automatisierung der kontinuierlichen Integration/kontinuierlichen Bereitstellung (CI/CD). So ist gewährleistet, dass Quellcodeverwaltung und Produktion übereinstimmen. Obwohl eine vollständige CI/CD-Pipeline für eine einfache Anwendung übertrieben erscheinen mag, ist die Automatisierung an sich, unabhängig von der Zuverlässigkeit, die sie bietet, in der Regel den Zeitaufwand für ihre Einrichtung wert. Außerdem lässt sich CI/CD im Nachhinein nur sehr schwer in eine bestehende, imperativ bereitgestellte Anwendung einbauen.

Auf diese mit YAML erstellte Anwendungsbeschreibung werden wir in späteren Abschnitten zurückkommen, um weitere Elemente wie die ConfigMap und Secrets-Volumes sowie die Pod Quality of Service zu untersuchen.

1.4 Externen Ingress für HTTP-Verkehr einrichten

Die Container für unsere Anwendung sind zwar jetzt bereitgestellt, aber es ist derzeit noch nicht möglich, dass jemand auf die Anwendung zugreift. Standardmäßig sind die Cluster-Ressourcen nur innerhalb des Clusters selbst verfügbar. Um unsere Anwendung der Außenwelt zugänglich zu machen, müssen wir einen Service und einen Load Balancer erstellen, um eine externe IP-Adresse bereitzustellen und den Datenverkehr zu unseren Containern zu leiten. Für die externe Bereitstellung werden wir zwei Kubernetes-Ressourcen verwenden.

Die erste Ressource ist ein Service, der für einen Lastausgleich des Datenverkehrs des Transmission Control Protocol (TCP) oder des User Datagram Protocol (UDP) sorgt. In unserem Fall nutzen wir das TCP-Protokoll. Die zweite ist eine Ingress-Ressource, die einen HTTP(S)-Lastausgleich mit intelligentem Routing von Anfragen auf der Grundlage von HTTP-Pfaden und Hosts bietet. Bei einer einfachen Anwendung wie dieser werden Sie sich vielleicht fragen, warum wir die komplexere Ingress-Ressource verwenden, aber wie Sie in späteren Abschnitten sehen werden, werden selbst bei dieser einfachen Anwendung HTTP-Anfragen von zwei verschiedenen Diensten bedient. Darüber hinaus bietet ein Ingress an der Schnittstelle nach außen die Flexibilität, die für zukünftige Erweiterungen unseres Dienstes erforderlich ist.

Hinweis

Die Ingress-Ressource ist eine der älteren Ressourcen in Kubernetes, und im Laufe der Jahre wurden zahlreiche Probleme mit der Art und Weise, wie sie den HTTP-Zugriff auf Microservices modelliert, gelöst. Dies hat zur Entwicklung der Gateway-API für Kubernetes geführt. Die Gateway-API wurde als Erweiterung für Kubernetes entwickelt und erfordert in Ihrem Cluster die Installation zusätzlicher Komponenten. Wenn Sie feststellen, dass Ingress Ihre Anforderungen nicht erfüllt, sollten Sie in Erwägung ziehen, zur Gateway-API zu wechseln.

Bevor die Ingress-Ressource definiert werden kann, muss ein Kubernetes-Service vorhanden sein, auf den der Ingress verweisen kann. Wir werden Labels verwenden, um den Dienst zu den Pods zu leiten, die wir im vorherigen Abschnitt erstellt haben. Der Service ist wesentlich einfacher zu definieren als das Deployment und sieht wie folgt aus:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: frontend
  name: frontend
  namespace: default
spec:
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    app: frontend
  type: ClusterIP
```

Nachdem Sie den Service definiert haben, können Sie eine Ingress-Ressource definieren. Im Gegensatz zu Service-Ressourcen muss für Ingress im Cluster ein Ingress-Controller-Container ausgeführt werden. Sie haben die Wahl zwischen verschiedenen Implementierungen, die entweder von Ihrem Cloud-Provider angeboten oder mit Open-Source-Servern implementiert werden. Wenn Sie sich für die Installation eines Open-Source-Ingress-Anbieters entscheiden, ist es eine gute Idee, den Helm-Paketmanager (*https://helm.sh*) zu verwenden, um ihn zu installieren und zu warten. Die Ingress-Provider `nginx` oder `haproxy` sind beliebte Optionen:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: frontend-ingress
spec:
  rules:
  - http:
      paths:
      - path: /testpath
        pathType: Prefix
        backend:
          service:
            name: test
            port:
              number: 8080
```

Nachdem unsere Ingress-Ressource erstellt wurde, ist unsere Anwendung bereit, den Datenverkehr von Webbrowsern aus der ganzen Welt zu bedienen. Als Nächstes werden wir uns ansehen, wie Sie Ihre Anwendung für eine einfache Konfiguration und Anpassung einrichten können.

1.5 Anwendung mit ConfigMaps konfigurieren

Jede Anwendung benötigt ein gewisses Maß an Konfiguration. Dabei kann es sich um die Anzahl der pro Seite anzuzeigenden Journaleinträge, die Farbe eines bestimmten Hintergrunds, eine spezielle Darstellung bei Feiertagen oder viele andere Arten der Konfiguration handeln. In der Regel ist es am besten, solche Konfigurationsinformationen von der Anwendung selbst zu trennen.

Für diese Trennung gibt es mehrere gute Gründe. Der erste Grund ist, dass Sie vielleicht dieselbe Binärdatei der Anwendung je nach Umgebung unterschiedlich konfigurieren möchten. In Europa möchten Sie vielleicht ein Osterspecial hervorheben, während Sie in China vielleicht ein Special zum chinesischen Neujahr anzeigen möchten. Neben dieser Spezialisierung auf bestimmte Umgebungen gibt es auch im Hinblick auf die Agilität Gründe für diese Trennung. Wenn Sie diese Funktionen per Code aktivieren, besteht die einzige Möglichkeit, die aktiven Funktionen zu ändern, darin, eine neue Binärdatei zu erstellen und zu veröffentlichen, was ein teurer und langsamer Prozess sein kann.

Die Verwendung der Konfiguration zur Aktivierung einer Reihe von Funktionen bedeutet, dass Sie Funktionen schnell (und sogar dynamisch) aktivieren und deaktivieren können, um auf Benutzeranforderungen oder Fehler im Anwendungscode zu reagieren. Features können so einzeln ein- und wieder ausgeschaltet werden. Diese Flexibilität stellt sicher, dass Sie mit den meisten Features kontinuierlich Fortschritte machen, selbst wenn einige zurückgesetzt werden müssen, um Leistungsprobleme oder Fehler zu beheben.

In Kubernetes wird diese Art von Konfiguration durch eine Ressource namens ConfigMap repräsentiert. Eine ConfigMap enthält mehrere Schlüssel/Wert-Paare, die Konfigurationsinformationen oder eine Datei darstellen. Diese Konfigurationsinformationen können einem Container in einem Pod entweder über Dateien oder Umgebungsvariablen zur Verfügung gestellt werden. Stellen Sie sich vor, Sie möchten, dass Ihre Online-Journalanwendung eine konfigurierbare Anzahl von Journaleinträgen pro Seite anzeigt. Um dies zu erreichen, können Sie eine ConfigMap wie folgt definieren:

```
kubectl create configmap frontend-config --from-literal=journalEntries=10
```

Um Ihre Anwendung zu konfigurieren, legen Sie die Konfigurationsinformationen als Umgebungsvariable in der Anwendung selbst offen. Dazu können Sie der Container-Ressource im Deployment, das Sie zuvor definiert haben, Folgendes hinzufügen:

```
...
# Das Array Containers im PodTemplate innerhalb des Deployments
containers:
  - name: frontend
    ...
    env:
    - name: JOURNAL_ENTRIES
      valueFrom:
        configMapKeyRef:
          name: frontend-config
          key: journalEntries
...
```

Obwohl hier gezeigt wird, wie Sie eine ConfigMap zur Konfiguration Ihrer Anwendung verwenden können, werden Sie in der realen Welt von Deployments regelmäßige Änderungen an dieser Konfiguration vornehmen wollen, zumindest wöchentlich. Es mag verlockend sein, diese Änderungen einfach in der ConfigMap selbst vorzunehmen, aber das ist aus verschiedenen Gründen keine Best Practice: Erstens löst eine Änderung der Konfiguration keine Aktualisierung der bestehenden Pods aus. Die Konfiguration wird erst beim Neustart des Pods übernommen. Daher ist der Rollout nicht auf bestimmte Dienstprobleme hin ausgerichtet und kann ad hoc oder zufällig erfolgen. Ein weiterer Grund ist, dass die einzige Versionierung für die ConfigMap in Ihrer Versionsverwaltung liegt und es sehr schwierig sein kann, ein Rollback durchzuführen.

Ein besserer Ansatz ist es, in den Namen der ConfigMap eine Versionsnummer aufzunehmen. Anstatt die Map `frontend-config` zu nennen, nennen Sie sie `frontend-config-v1`. Wenn Sie eine Änderung vornehmen möchten, erstellen Sie eine neue ConfigMap `v2` und aktualisieren die Deployment-Ressource, um diese Konfiguration zu verwenden, anstatt die ConfigMap selbst zu aktualisieren. Wenn Sie dies tun, wird automatisch ein Deployment-Rollout ausgelöst, bei dem die entsprechenden Zustandsprüfungen und Pausen zwischen den Änderungen durchgeführt werden. Sollten Sie jemals ein Rollback durchführen müssen, befindet sich die `v1`-Konfiguration im Cluster, und das Rollback ist einfach, da Sie dann lediglich eine erneute Aktualisierung des Deployments durchführen müssen.

1.6 Authentifizierung mit Secrets verwalten

Bisher ging es noch nicht wirklich um den Redis-Dienst, mit dem unser Frontend verbunden ist. Aber in jeder echten Anwendung müssen wir die Verbindungen zwischen unseren Diensten sichern. Zum einen, um den Schutz der Benutzer und ihrer Daten zu gewährleisten, und zum anderen ist es wichtig, Fehler zu vermeiden, wie z. B. ein Entwicklungs-Frontend mit einer Produktivdatenbank zu verbinden.

Die Redis-Datenbank wird mit einem einfachen Passwort authentifiziert. Es mag bequem erscheinen, dieses Passwort im Quellcode Ihrer Anwendung oder in einer Datei in Ihrem Image zu speichern, aber beides ist aus verschiedenen Gründen keine gute Idee. Der erste Grund ist, dass Sie Ihr Secret (das Kennwort) in einer Umgebung öffentlich zugänglich machen, in der Sie nicht unbedingt an Zugriffssteuerung denken. Wenn Sie ein Kennwort in Ihre Quellcodeverwaltung einbauen, dann hat jeder, der Zugriff auf Ihren Quellcode hat, gleichzeitig auch Zugriff auf alle Secrets. Das ist nicht die beste Vorgehensweise, da Sie wahrscheinlich eine größere Anzahl von Benutzern haben werden, die auf Ihren Quellcode zugreifen können, als die, die Zugriff auf Ihre Redis-Instanz haben sollten. Außerdem sollte jemand, der Zugriff auf Ihr Container-Image hat, nicht unbedingt Zugriff auf Ihre Produktivdatenbank haben.

Neben den Bedenken hinsichtlich der Zugriffskontrolle ist ein weiterer Grund, Secrets nicht an die Versionsverwaltung und/oder Images zu binden, die Parametrisierung. Sie möchten in der Lage sein, denselben Quellcode und dieselben Images in ver-

schiedenen Umgebungen (z. B. Entwicklung, Canary und Produktion) zu verwenden. Wenn die Secrets im Quellcode oder in einem Image fest verdrahtet sind, benötigen Sie für jede Umgebung ein anderes Image (oder einen anderen Code).

Nachdem Sie im vorherigen Abschnitt die ConfigMaps kennengelernt haben, könnten Sie sofort denken, dass das Kennwort als Konfiguration gespeichert und dann als anwendungsspezifische Konfiguration in die Anwendung eingefügt werden könnte. Sie haben völlig recht, wenn Sie glauben, dass die Trennung von Konfiguration und Anwendung dasselbe ist wie die Trennung von Secrets und Anwendung. Aber in Wahrheit ist ein Secret ein wichtiges Konzept für sich. Wahrscheinlich möchten Sie die Zugriffskontrolle, die Handhabung und die Aktualisierung von Secrets anders handhaben als die einer Konfiguration. Noch wichtiger ist, dass Sie möchten, dass Ihre Entwickler beim Zugriff auf Secrets anders denken als beim Zugriff auf Konfigurationen. Aus diesen Gründen verfügt Kubernetes über eine integrierte Secret-Ressource für die Verwaltung von geheimen, vertraulichen Daten.

Sie können für Ihre Redis-Datenbank wie folgt ein geheimes Kennwort erstellen:

```
kubectl create secret generic redis-passwd --from-literal=passwd=${RANDOM}
```

Es liegt auf der Hand, dass Sie für Ihr Kennwort etwas anderes als eine Zufallszahl nehmen möchten. Außerdem möchten Sie wahrscheinlich einen Service zur Verwaltung von Geheimnissen/Schlüsseln einsetzen, entweder über Ihren Cloud-Anbieter, wie Microsoft Azure Key Vault, oder ein Open-Source-Projekt, wie Vault von HashiCorp. Ein Schlüsselverwaltungsdienst verfügt in der Regel über eine engere Integration mit Kubernetes-Secrets.

Nachdem Sie das Redis-Kennwort als Secret in Kubernetes gespeichert haben, müssen Sie dieses Secret an die laufende Anwendung binden, wenn diese in Kubernetes bereitgestellt wird. Hierfür können Sie ein Kubernetes-Volume verwenden. Ein Volume ist eine Datei oder ein Verzeichnis, das in einen laufenden Container an einem benutzerdefinierten Ort eingebunden werden kann. Im Falle von Secrets wird das Volume als tmpfs-RAM-gestütztes Dateisystem erstellt und dann in den Container eingehängt. Dadurch ist es für einen Angreifer sehr viel schwieriger, an die Secrets heranzukommen, selbst wenn der Rechner physisch kompromittiert wird (was in der Cloud recht unwahrscheinlich, im Rechenzentrum jedoch möglich ist).

Hinweis

In Kubernetes werden Secrets standardmäßig unverschlüsselt gespeichert. Wenn Sie Secrets verschlüsselt speichern möchten, können Sie mit einem Schlüsselanbieter zusammenarbeiten, der Ihnen einen Schlüssel liefert, den Kubernetes zur Verschlüsselung aller Secrets im Cluster verwendet. Beachten Sie, dass dies zwar die Schlüssel gegen direkte Angriffe auf die `etcd`-Datenbank schützt, der Zugriff über den Kubernetes-API-Server aber dennoch ordnungsgemäß gesichert sein muss.

Um einem Deployment ein Volume für Secrets hinzuzufügen, müssen Sie zwei neue Einträge in der YAML für das Deployment angeben. Der erste ist ein Volume-Eintrag für den Pod, der das Volume zum Pod hinzufügt:

```
...
  volumes:
  - name: passwd-volume
    secret:
      secretName: redis-passwd
```

Mit CSI-Treibern (Container Storage Interface) können Sie Schlüsselverwaltungssysteme (Key Management System, KMS) verwenden, die sich außerhalb Ihres Kubernetes-Clusters befinden. Dies ist häufig eine Anforderung für die Einhaltung von Vorschriften und die Sicherheit in großen oder regulierten Organisationen. Wenn Sie einen dieser CSI-Treiber verwenden, würde Ihr Volume stattdessen wie folgt aussehen:

```
...
  volumes:
  - name: passwd-volume
    csi:
      driver: secrets-store.csi.k8s.io
      readOnly: true
      volumeAttributes:
        secretProviderClass: "azure-sync"
...
```

Unabhängig von der Methode müssen Sie das im Pod definierte Volume in einen bestimmten Container mounten. Dies geschieht über das Feld `volumeMounts` in der Container-Beschreibung:

```
...
  volumeMounts:
  - name: passwd-volume
    readOnly: true
    mountPath: "/etc/redis-passwd"
...
```

Dadurch wird das Volume mit den Secrets in das Verzeichnis `redis-passwd` eingebunden und der Client-Code kann dann darauf zugreifen. Baut man dies alles zusammen, erhält man das folgende vollständige Deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: frontend
  name: frontend
  namespace: default
spec:
  replicas: 2
  selector:
```

```
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - image: my-repo/journal-server:v1-abcde
        imagePullPolicy: IfNotPresent
        name: frontend
        volumeMounts:
        - name: passwd-volume
          readOnly: true
          mountPath: "/etc/redis-passwd"
        resources:
          request:
            cpu: "1.0"
            memory: "1G"
          limits:
            cpu: "1.0"
            memory: "1G"
      volumes:
      - name: passwd-volume
        secret:
          secretName: redis-passwd
```

An diesem Punkt haben wir die Clientanwendung so konfiguriert, dass ihr für die Authentifizierung gegenüber dem Redis-Dienst ein Secret zur Verfügung steht. Die Konfiguration von Redis für die Verwendung dieses Kennworts ist ähnlich; wir binden es in den Redis-Pod ein und laden das Passwort aus der Datei.

1.7 Einfache zustandsbehaftete Datenbank bereitstellen

Obwohl die Bereitstellung einer zustandsbehafteten Anwendung konzeptionell ähnlich ist wie die Bereitstellung eines Clients wie unseres Frontends, macht die Tatsache, dass wir es mit einem zustandsbehafteten Service zu tun haben, die Dinge komplizierter. Die erste Komplikation besteht darin, dass in Kubernetes ein Pod aus verschiedenen Gründen verschoben werden kann, z. B. wegen des Zustands des Knotens, eines Upgrades oder wenn die Last neu verteilt wird. In diesem Fall wird der Pod möglicherweise auf eine andere Maschine verschoben. Wenn sich die mit der Redis-Instanz verbundenen Daten auf einer bestimmten Maschine oder im Container selbst befinden, gehen diese Daten verloren, wenn der Container migriert oder neu gestartet wird. Um dies zu verhindern, ist es bei der Ausführung zustandsbehafteter Workloads in Kubernetes wichtig, entfernte *persistente Volumes* zu verwenden, um den mit der Anwendung verbundenen Zustand zu verwalten.

Es gibt eine Vielzahl von Implementierungen von PersistentVolumes-Objekten in Kubernetes, aber alle besitzen gemeinsame Merkmale. Wie die zuvor beschriebenen Secret-Volumes sind sie mit einem Pod verbunden und werden an einem bestimmten Ort in einen Container eingebunden. Im Gegensatz zu Secrets handelt es sich bei PersistentVolumes in der Regel um Remote-Speicher, der über eine Art Netzwerkprotokoll eingebunden wird, entweder dateibasiert (z. B. Network File System, NFS) oder Server Message Block (SMB) oder blockbasiert (iSCSI, cloudbasierte Festplatten usw.). Für Anwendungen wie Datenbanken sind blockbasierte Festplatten in der Regel vorzuziehen, da sie eine bessere Leistung bieten, aber wenn die Leistung weniger wichtig ist, bieten dateibasierte Datenträger manchmal eine größere Flexibilität.

Hinweis

Die Verwaltung von Zuständen ist im Allgemeinen kompliziert, und Kubernetes ist da keine Ausnahme. Wenn Sie in einer Umgebung arbeiten, die zustandsbehaftete Dienste unterstützt (z.B. MySQL as a Service, Redis as a Service), ist es im Allgemeinen eine gute Idee, diese zustandsbehafteten Dienste zu verwenden. Auf den ersten Blick mögen die Mehrkosten für zustandsbehaftete Software as a Service (SaaS) hoch erscheinen, aber wenn man alle betrieblichen Anforderungen des Zustands (Backup, Datenlokalisierung, Redundanz usw.) und die Tatsache berücksichtigt, dass das Vorhandensein des Zustands in einem Kubernetes-Cluster das Verschieben von Anwendungen zwischen Clustern erschwert, wird klar, dass Storage-SaaS in den meisten Fällen den Preisaufschlag wert ist. In lokalen Umgebungen, in denen Storage-SaaS nicht verfügbar ist, ist es definitiv besser, ein spezielles Team mit der Bereitstellung von Storage als Service für das gesamte Unternehmen zu beauftragen, als jedem Team zu erlauben, es selbst aufzubauen.

Für die Bereitstellung unseres Redis-Dienstes verwenden wir eine StatefulSet-Ressource. StatefulSets wurden nach der ersten Kubernetes-Version als Ergänzung zu ReplicaSet-Ressourcen hinzugefügt und bieten etwas bessere Garantien wie konsistente Namen (keine zufälligen Hashes!) und eine definierte Reihenfolge für das Hochskalieren und Herunterskalieren. Wenn Sie ein Singleton bereitstellen, ist dies weniger wichtig, aber wenn Sie einen replizierten Zustand bereitstellen möchten, sind diese Attribute sehr praktisch.

Um ein PersistentVolume für unser Redis zu erhalten, verwenden wir einen PersistentVolumeClaim. Man kann sich einen Claim als eine »Anfrage nach Ressourcen« vorstellen. Unser Redis erklärt abstrakt, dass er 50 GB Speicherplatz benötigt, und der Kubernetes-Cluster bestimmt, wie er ein entsprechendes PersistentVolume bereitstellen kann. Hierfür gibt es zwei Gründe. Der erste ist, dass wir ein StatefulSet schreiben können, das zwischen verschiedenen Clouds und vor Ort portabel ist, wo die Details der Festplatten unterschiedlich sein können. Der andere Grund ist, dass viele PersistentVolume-Typen zwar nur auf einen einzigen Pod gemountet werden können, wir aber mithilfe von Volume-Claims eine Vorlage schreiben können, die repliziert werden kann und dennoch jedem Pod sein eigenes spezifisches PersistentVolume zuweist.

Das folgende Beispiel zeigt ein Redis StatefulSet mit PersistentVolumes:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
spec:
  serviceName: "redis"
  replicas: 1
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: redis
        image: redis:5-alpine
        ports:
        - containerPort: 6379
          name: redis
        volumeMounts:
        - name: data
          mountPath: /data
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 10Gi
```

Damit wird eine einzelne Instanz Ihres Redis-Dienstes bereitgestellt. Nehmen Sie jedoch an, dass Sie den Redis-Cluster replizieren möchten, um Lesevorgänge zu skalieren und Ausfallsicherheit zu gewährleisten. Dazu müssen Sie natürlich die Anzahl der Replikate auf drei erhöhen, aber Sie müssen auch sicherstellen, dass die beiden neuen Replikate mit dem Schreib-Master für Redis verbunden sind. Wie Sie diese Verbindung herstellen, erfahren Sie im folgenden Abschnitt.

Wenn Sie den headless Service für das Redis StatefulSet erstellen, wird ein DNS-Eintrag `redis-0.redis` erstellt; dies ist die IP-Adresse des ersten Replikats. Damit können Sie ein einfaches Skript erstellen, das in allen Containern gestartet werden kann:

```
!/bin/sh

PASSWORD=$(cat /etc/redis-passwd/passwd)

if [[ "${HOSTNAME}" == "redis-0" ]]; then
  redis-server --requirepass ${PASSWORD}
else
```

```
  redis-server --slaveof redis-0.redis 6379 --masterauth ${PASSWORD}
    --requirepass ${PASSWORD}
fi
```

Sie können dieses Skript als ConfigMap erstellen:

```
kubectl create configmap redis-config --from-file=./launch.sh
```

Diese ConfigMap fügen Sie dann zu Ihrem StatefulSet hinzu und nutzen sie als Befehl für den Container. Lassen Sie uns außerdem das Kennwort für die Authentifizierung hinzufügen, das wir weiter vorne im Kapitel erstellt haben.

Das vollständige Redis-System mit drei Replikaten sieht wie folgt aus:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
name: redis
spec:
  serviceName: "redis"
  replicas: 3
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: redis
        image: redis:5-alpine
        ports:
        - containerPort: 6379
          name: redis
        volumeMounts:
        - name: data
          mountPath: /data
        - name: script
          mountPath: /script/launch.sh
          subPath: launch.sh
        - name: passwd-volume
          mountPath: /etc/redis-passwd
        command:
        - sh
        - -c
        - /script/launch.sh
      volumes:
      - name: script
        configMap:
          name: redis-config
          defaultMode: 0777
      - name: passwd-volume
        secret:
          secretName: redis-passwd
```

```
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 10Gi
```

Jetzt ist Ihr Redis für Fehlertoleranz geclustert. Wenn eines der drei Redis-Replikate aus irgendeinem Grund ausfällt, kann Ihre Anwendung mit den beiden verbleibenden Replikaten weiterlaufen, bis das dritte Replikat wiederhergestellt ist.

1.8 TCP-Load-Balancer mithilfe von Services erstellen

Nachdem wir den zustandsbehafteten Redis-Service bereitgestellt haben, müssen wir ihn für unser Frontend verfügbar machen. Zu diesem Zweck erstellen wir zwei verschiedene Kubernetes-Services. Der erste ist der Service, der Daten aus Redis liest. Da Redis die Daten an alle drei Mitglieder des StatefulSet repliziert, ist es uns egal, an welches Replikat unsere Leseanfrage geht. Daher verwenden wir für die Lesevorgänge einen Basis-Service:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: redis
  name: redis
  namespace: default
spec:
  ports:
  - port: 6379
    protocol: TCP
    targetPort: 6379
  selector:
    app: redis
  sessionAffinity: None
  type: ClusterIP
```

Um Schreibvorgänge zu ermöglichen, müssen Sie den Redis-Master (Replikat Nr. 0) anvisieren. Dazu erstellen Sie einen *headless Service*. Ein headless Service hat keine Cluster-IP-Adresse, sondern programmiert einen DNS-Eintrag für jeden Pod im StatefulSet. Das bedeutet, dass wir auf unseren Master über den DNS-Namen `redis-0.redis` zugreifen können:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: redis-write
```

```
    name: redis-write
  spec:
    clusterIP: None
    ports:
    - port: 6379
    selector:
      app: redis
```

Wenn wir uns also mit Redis für Schreibvorgänge oder transaktionale Lese-/Schreibpaare verbinden wollen, können wir einen separaten Schreib-Client erstellen, der mit dem `redis-0.redis-write`-Server verbunden ist.

1.9 Ingress zur Weiterleitung des Datenverkehrs an einen statischen Dateiserver verwenden

Die letzte Komponente unserer Anwendung ist ein *statischer Dateiserver*. Der Server für statische Dateien ist für die Bereitstellung von HTML-, CSS-, JavaScript- und Bilddateien zuständig. Es ist sowohl effizienter als auch zielgerichteter für uns, die Bereitstellung statischer Dateien von unserem zuvor beschriebenen API-Serving-Frontend zu trennen. Wir können problemlos einen hochleistungsfähigen statischen Standarddateiserver wie NGINX für die Bereitstellung von Dateien verwenden, während sich unsere Entwicklungsteams auf den für die Implementierung unserer API erforderlichen Code konzentrieren können.

Glücklicherweise macht die Ingress-Ressource diese Art von Mini-Microservice-Architektur sehr einfach. Genau wie beim Frontend können wir eine Deployment-Ressource verwenden, um einen replizierten NGINX-Server zu beschreiben. Lassen Sie uns die statischen Images in den NGINX-Container einbauen und diese an jedes Replikat verteilen. Die Deployment-Ressource sieht wie folgt aus:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: fileserver
  name: fileserver
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: fileserver
  template:
    metadata:
      labels:
        app: fileserver
    spec:
      containers:
```

```
        # Dieses Image ist als Beispiel gedacht, ersetzen Sie es durch Ihre eigene
        # statische Image-Datei.
        - image: my-repo/static-files:v1-abcde
          imagePullPolicy: Always
          name: fileserver
          terminationMessagePath: /dev/termination-log
          terminationMessagePolicy: File
          resources:
            requests:
              cpu: "1.0"
              memory: "1G"
            limits:
              cpu: "1.0"
              memory: "1G"
      dnsPolicy: ClusterFirst
      restartPolicy: Always
```

Nachdem nun ein replizierter statischer Webserver eingerichtet und in Betrieb ist, erstellen Sie ebenfalls eine Service-Ressource, die als Lastausgleich fungiert:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: fileserver
  name: fileserver
  namespace: default
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: fileserver
  sessionAffinity: None
  type: ClusterIP
```

Da Sie nun einen Service für Ihren statischen Dateiserver haben, erweitern Sie die Ingress-Ressource um den neuen Pfad. Es ist wichtig, dass Sie den Pfad / nach dem Pfad /api platzieren, sonst würde er /api subsumieren und API-Anfragen an den statischen Dateiserver leiten. Der neue Ingress sieht wie folgt aus:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: frontend-ingress
spec:
  rules:
  - http:
      paths:
      - path: /api
        pathType: Prefix
        backend:
```

```
      service:
        name: fileserver
        port:
          number: 8080
    # HINWEIS: Dies sollte nach /api kommen, da es sonst Anfragen abfängt.
    - path: /
      pathType: Prefix
      backend:
        service:
          name: fileserver
          port:
            number: 80
```

Da Sie nun, zusätzlich zum Ingress für die API, eine Ingress-Ressource für Ihren Dateiserver eingerichtet haben, ist die Benutzeroberfläche der Anwendung einsatzbereit. Die meisten modernen Anwendungen kombinieren statische Dateien, normalerweise HTML und JavaScript, mit einem dynamischen API-Server, der in einer serverseitigen Programmiersprache wie Java, .NET oder Go implementiert ist.

1.10 Ihre Anwendung mit Helm parametrisieren

Alles, was wir bisher beschrieben haben, konzentriert sich auf die Bereitstellung einer einzelnen Instanz unseres Dienstes in einem einzelnen Cluster. In der Realität wird jedoch fast jeder Service und jedes Serviceteam in mehreren Umgebungen bereitgestellt werden müssen (selbst wenn sie sich einen Cluster teilen). Selbst wenn Sie nur ein einzelner Entwickler sind, der an einer einzigen Anwendung arbeitet, möchten Sie wahrscheinlich mindestens eine Entwicklungs- und eine Produktivversion Ihrer Anwendung haben, damit Sie iterieren und entwickeln können, ohne die Nutzer der Produktivumgebung zu beeinträchtigen. Wenn Sie Integrationstests und CI/CD in Betracht ziehen, ist es wahrscheinlich, dass Sie selbst bei einem einzigen Service und einer Handvoll Entwickler mindestens drei verschiedene Umgebungen bereitstellen möchten, möglicherweise sogar mehr, wenn Sie den Umgang mit Ausfällen auf Rechenzentrumsebene berücksichtigen. Lassen Sie uns ein paar Optionen für die Bereitstellung untersuchen.

Ein anfänglicher Fehler vieler Teams besteht darin, die Dateien einfach von einem Cluster in einen anderen zu kopieren. Statt eines einzigen Verzeichnisses *frontend/* gibt es dann zwei Verzeichnisse: *frontend-production/* und *frontend-development/*. Dies ist zwar eine praktikable Option, aber auch gefährlich, da Sie nun dafür sorgen müssen, dass diese Dateien miteinander synchronisiert bleiben. Wenn sie völlig identisch sein sollten, wäre dies einfach, aber es ist zu erwarten, dass es zu einer gewissen Abweichung zwischen Entwicklung und Produktion kommt, da Sie neue Funktionen entwickeln werden. Es ist von entscheidender Bedeutung, dass diese Abweichung sowohl beabsichtigt als auch leicht zu handhaben ist.

Eine andere Möglichkeit, dies zu erreichen, wäre die Verwendung von Zweigen und Versionsverwaltungssystemen, wobei die Produktions- und Entwicklungszweige von einem zentralen Repository ausgehen und die Unterschiede zwischen den Zwei-

gen deutlich sichtbar sind. Dies kann für einige Teams eine praktikable Option sein, aber die Mechanik des Wechsels zwischen Zweigen ist eine Herausforderung, wenn Sie Software gleichzeitig in verschiedenen Umgebungen bereitstellen möchten (z. B. ein CI/CD-System, das in verschiedenen Cloud-Regionen bereitgestellt wird).

Daher verwenden die meisten Anwender ein *Templating-System*. Ein Templating-System kombiniert Templates, die das zentrale Rückgrat der Anwendungskonfiguration bilden, mit Parametern, die das Template auf eine bestimmte Umgebungskonfiguration *spezialisieren*. Auf diese Weise können Sie eine allgemeine, gemeinsam genutzte Konfiguration nutzen, mit absichtlichen (und leicht verständlichen) Anpassungen, die Sie nach Bedarf vornehmen. Es gibt eine Vielzahl von Vorlagensystemen für Kubernetes, aber das mit Abstand beliebteste ist Helm.

In Helm wird eine Anwendung in eine Sammlung von Dateien verpackt, die als *Chart* (Seekarte) bezeichnet wird (in der Welt der Container und Kubernetes gibt es viele nautische Witze).

Ein Chart beginnt mit einer *chart.yaml*-Datei, die die Metadaten für das Diagramm selbst definiert:

```
apiVersion: v1
appVersion: "1.0"
description: Ein Helm-Chart für unseren Frontend-Journal-Server.
name: frontend
version: 0.1.0
```

Diese Datei wird im Stammverzeichnis des Charts abgelegt (z. B. *frontend/*). Innerhalb dieses Verzeichnisses gibt es ein Verzeichnis *templates*, in dem die Templates abgelegt werden. Ein Template ist im Grunde eine YAML-Datei aus den vorangegangenen Beispielen, wobei einige der Werte in der Datei durch Parameterreferenzen ersetzt werden. Stellen Sie sich zum Beispiel vor, dass Sie die Anzahl der Replikate in Ihrem Frontend parametrisieren möchten. Zuvor sah das Deployment folgendermaßen aus:

```
...
spec:
  replicas: 2
...
```

In der Vorlagendatei *(frontend-deployment.tmpl)* sieht es stattdessen wie folgt aus:

```
...
spec:
  replicas: {{ .replicaCount }}
...
```

Das bedeutet, dass Sie bei der Bereitstellung des Charts den Wert für Replikate durch den entsprechenden Parameter ersetzen. Die Parameter selbst werden in einer *values.yaml*-Datei definiert. Für jede Umgebung, in der die Anwendung bereitgestellt werden soll, wird eine Wertedatei erstellt. Die Wertedatei für dieses einfache Chart würde wie folgt aussehen:

```
replicaCount: 2
```

Wenn Sie alles zusammennehmen, können Sie dieses Chart mit dem Helm-Tool folgendermaßen bereitstellen:

```
helm install path/to/chart --values path/to/environment/values.yaml
```

Hierdurch wird Ihre Anwendung parametrisiert und in Kubernetes bereitgestellt. Im Laufe der Zeit wird die Anzahl der Parametrisierungen zunehmen, um so die Vielfalt der Umgebungen für Ihre Anwendung zu erfassen.

1.11 Best Practices für die Bereitstellung von Services

Kubernetes ist ein leistungsstarkes System, das komplex erscheinen kann. Das Einrichten einer grundlegenden Anwendung kann jedoch sehr einfach werden, wenn Sie sich an die folgenden Best Practices halten:

- Die meisten Services sollten als Deployment-Ressourcen bereitgestellt werden. Deployments erstellen identische Replikate für Redundanz und Skalierung.
- Deployments können über einen Service zur Verfügung gestellt werden, bei dem es sich um einen Load Balancer handelt. Ein Service kann entweder innerhalb eines Clusters (Standard) oder extern bereitgestellt werden. Wenn Sie eine HTTP-Anwendung bereitstellen möchten, können Sie einen Ingress-Controller verwenden, um Dinge wie Request Routing und SSL hinzuzufügen.
- Letztendlich werden Sie Ihre Anwendung parametrisieren wollen, um ihre Konfiguration in verschiedenen Umgebungen wiederverwendbar zu machen. Paketmanager wie Helm sind die beste Wahl für diese Art der Parametrisierung.

1.12 Zusammenfassung

Die in diesem Kapitel erstellte Anwendung ist einfach, enthält aber fast alle Konzepte, die Sie für die Erstellung größerer, komplizierterer Anwendungen benötigen. Um erfolgreich mit Kubernetes arbeiten zu können, ist es wichtig zu verstehen, wie die einzelnen Teile zusammenpassen und wie man die grundlegenden Kubernetes-Komponenten verwendet.

Die Schaffung der richtigen Grundlage durch Versionsverwaltung, Code-Review und kontinuierliche Bereitstellung Ihres Service stellt sicher, dass das, was Sie entwickeln, solide gebaut ist. Wenn wir in den folgenden Kapiteln die fortgeschritteneren Themen durchgehen, sollten Sie diese grundlegenden Informationen im Hinterkopf behalten.