KAPITEL 3

Declarative Deployment

Das Herz des Pattern *Declarative Deployment* ist die Deployment-Ressource von Kubernetes. Diese Abstrahierung kapselt den Upgrade- und Rollback-Prozess einer Gruppe von Containern und macht dessen Ausführung zu einer wiederholbaren und automatisierten Aktion.

Problem

Sie können per »Selfservice« isolierte Umgebungen als Namensräume anlegen und die Services in diesen Umgebungen mit minimalem manuellem Eingriff durch den Scheduler platzieren lassen. Aber mit einer wachsenden Zahl von Microservices wird deren Aktualisierung und Ersetzen durch neuere Versionen zunehmend zu einer Belastung.

Zum Aktualisieren eines Service auf die nächste Version gehören Dinge wie das Starten der neuen Version des Pods, das »elegante« Stoppen der alten Version auf einem Pod, das Warten und Verifizieren, dass der Start erfolgreich war, und manchmal auch das Zurückrollen all dessen zur vorherigen Version, wenn es zu einem Fehler kam. Diese Aktivitäten werden entweder durchgeführt, indem Downtime zugelassen wird, dafür dann aber keine konkurrierenden Service-Versionen laufen, oder ohne Downtime, dafür aber mit einem gestiegenen Ressourcen-Bedarf, weil beide Versionen des Service während des Update-Prozesses laufen. Das manuelle Ausführen dieser Schritte kann zu Fehlern durch die Person führen, und das Erstellen sauberer Skripte für den Prozess kann viel Aufwand bedeuten – was den Release-Prozess in beiden Fällen schnell zum Flaschenhals machen kann.

Lösung

Zum Glück hat Kubernetes diese Aktivität ebenfalls automatisiert. Mit dem Konzept des *Deployments* können Sie beschreiben, wie Ihre Anwendung aktualisiert werden soll. Dabei stehen verschiedene Strategien zur Verfügung und die einzelnen Aspekte des Update-Prozesses lassen sich definieren. Wenn Sie überlegen, dass Sie für jede Microservice-Instanz pro Release-Zyklus mehrere Deployments durchfüh-

ren (was abhängig vom Team und Projekt von Minuten bis zu mehreren Monaten reichen kann), ist dies eine weitere Automatisierung durch Kubernetes, die Aufwand spart.

Imperative rollierende Updates mit kubectl sind veraltet

Kubernetes hat rollierende Updates von Anfang an unterstützt. Die erste Implementierung war im Kern *imperativ*, der `kubectl`-Client hat dem Server gesagt, was für jeden Update-Schritt zu tun ist.

Auch wenn der Befehl `kubectl rolling-update` weiterhin vorhanden ist, steht er definitiv auf der Abschussliste, denn es gibt bei solch einem imperativen Vorgehen eine Reihe von Nachteilen:

- Statt den gewünschten Endzustand zu beschreiben, feuert `kubectl rolling-update` Befehle, um das System in den gewünschten Zustand zu bringen.
- Die gesamte Orchestrierungs-Logik zum Ersetzen der Container und der ReplicationController werden durch `kubectl` durchgeführt, das während des Update-Prozesses hinter den Kulissen auf den API-Server lauscht und mit ihm interagiert, womit eine inhärent serverseitige Verantwortung auf den Client übertragen wird.
- Sie brauchen eventuell mehr als einen Befehl, um das System in den gewünschten Zustand zu bringen. Diese Befehle müssen automatisiert und in verschiedenen Umgebungen wiederholbar sein.
- Im Laufe der Zeit überschreibt eventuell jemand Ihre Änderungen.
- Der Update-Prozess muss dokumentiert und aktuell gehalten werden, während sich der Service weiterentwickelt.
- Die einzige Möglichkeit, herauszufinden, was Sie deployt haben, ist das Prüfen des Systemzustands. Manchmal befindet sich der Status des aktuellen Systems nicht im gewünschten Zustand – dann müssen Sie ihn mit der Deployment-Dokumentation in Übereinstimmung bringen.

Stattdessen wurde das Ressourcen-Objekt Deployment eingeführt, um ein *deklaratives Update* zu unterstützen, das vollständig vom Kubernetes-Backend verwaltet wird. Da deklarative Updates so viele Vorteile besitzen und die Unterstützung für imperative Updates irgendwann verschwinden wird, konzentrieren wir uns in diesem Pattern nur auf deklarative Updates.

In Kapitel 2 haben Sie gesehen, dass der Scheduler zum effektiven Erledigen seines Jobs ausreichend Ressourcen auf dem Host-System, passende Platzierungs-Richtlinien und Container mit adäquat definierten Ressourcen-Profilen benötigt. Genauso muss ein Deployment, um seine Aufgabe ordentlich zu erledigen, davon ausgehen können, dass die Container gute Cloud-native Mitbürger sind. Der Kern eines Deployments ist die Fähigkeit, einen Satz von Pods zuverlässig starten und stoppen zu können. Damit das wie gewünscht funktioniert, lauschen die Container normaler-

weise selbst auf Lifecycle-Events und reagieren darauf (zum Beispiel SIGTERM – siehe Kapitel 5); zudem bieten sie Health-Check-Endpunkte wie in Kapitel 4 beschrieben, über die man herausfindet, ob die Container erfolgreich gestartet wurden.

Deckt ein Container diese beiden Bereiche korrekt ab, kann die Plattform alte Container sauber herunterfahren und sie ersetzen, indem sie aktualisierte Instanzen startet. Dann können alle restlichen Aspekte eines Update-Prozesses deklarativ definiert und als eine atomare Aktion mit vordefinierten Schritten und einem erwarteten Ergebnis ausgeführt werden. Schauen wir uns die Optionen für das Update-Verhalten eines Containers an.

Rollierendes Deployment

Der deklarative Weg zum Aktualisieren von Anwendungen in Kubernetes nutzt das Konzept des Deployments. Unter der Motorhaube erzeugt das Deployment ein ReplicaSet, das Set-basierte Label-Selektoren unterstützt. Auch erlaubt die Deployment-Abstraktion das Anpassen des Verhaltens des Update-Prozesses durch Strategien wie `RollingUpdate` (der Standard) oder `Recreate`. In Beispiel 3-1 sehen Sie die wichtigsten Elemente zum Konfigurieren eines Deployments für ein rollierendes Update.

Beispiel 3-1: Deployment für ein rollierendes Update

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: random-generator
spec:
  replicas: 3  ❶
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1  ❷
      maxUnavailable: 1  ❸
    selector:
      matchLabels:
        app: random-generator
    template:
      metadata:
        labels:
          app: random-generator
      spec:
        containers:
        - image: k8spatterns/random-generator:1.0
          name: random-generator
          readinessProbe:  ❹
            exec:
              command: [ "stat", "/random-generator-ready" ]
```

❶ Deklaration von drei Replicas. Damit ein rollierendes Update sinnvoll ist, brauchen Sie mehr als eine Replica.

❷ Anzahl an Pods, die temporär während eines Updates zusätzlich zu den spezifizierten Replicas ausgeführt werden können. In diesem Beispiel können es so maximal vier Replicas sein.

❸ Anzahl an Pods, die während des Updates nicht zur Verfügung stehen müssen. Hier ist es möglich, dass während des Updates nur zwei Pods verfügbar sind.

❹ Die Readiness-Proben sind für ein rollierendes Deployment sehr wichtig, um Zero Downtime zu ermöglichen – vergessen Sie sie nicht (siehe Kapitel 4).

Die Strategie `RollingUpdate` stellt sicher, dass es während des Update-Prozesses zu keiner Downtime kommt. Hinter den Kulissen führt die Deployment-Implementierung gleiche Schritte durch, indem sie neue ReplicaSets erzeugt und alte Container durch neue ersetzt. Ein Vorteil beim Einsatz eines Deployment-Objekts ist, dass Sie die Rate für den Rollout neuer Container steuern können. Mit den Feldern `maxSurge` und `maxUnavailable` ist es Ihnen möglich, den Bereich der verfügbaren und zusätzlichen Pods anzugeben. In Abbildung 3-1 sehen Sie den Prozess dargestellt.

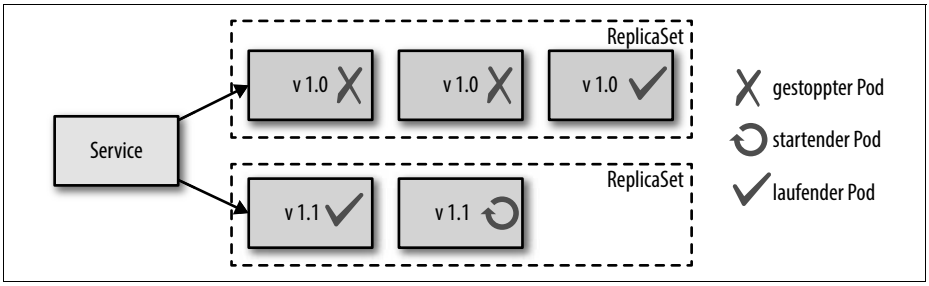

Abbildung 3-1: Rollierendes Deployment

Um ein deklaratives Update auszulösen, haben Sie drei Optionen:

- Ersetzen Sie das komplette Deployment durch die neue Version des Deployments mit `kubectl replace`.
- Patchen (`kubectl patch`) oder editieren Sie das Deployment interaktiv (`kubectl edit`), um das neue Container-Image der neuen Version festzulegen.
- Nutzen Sie `kubectl set image`, um das neue Image im Deployment zu setzen.

Schauen Sie sich auch das vollständige Beispiel in unserem Repository an *(http://bit.ly/2Fc6d6J)*, in dem der Einsatz dieser Befehle gezeigt wird, und Sie sehen, wie Sie ein Upgrade mit `kubectl rollout` überwachen und monitoren können.

Neben dem Vermeiden der schon erwähnten Nachteile des imperativen Deployens von Services erhalten Sie mit *Deployment* noch weitere Vorteile:

- Deployment ist eine Kubernetes-Ressource, deren Status vollständig intern von Kubernetes verwaltet wird. Der gesamte Update-Prozess wird ohne clientseitige Interaktion auf dem Server durchgeführt.

- Die deklarative Natur von Deployments ermöglicht es Ihnen, zu erkennen, wie der deployte Status aussehen soll, statt die Schritte zu beschreiben, die notwendig sind, um dorthin zu gelangen.
- Die Deployment-Definition ist ein ausführbares Objekt, das in vielen Umgebungen ausprobiert und getestet wurde, bevor es produktiv ging.
- Der Update-Prozess wird zudem vollständig aufgezeichnet und versioniert. Sie haben die Möglichkeit, ihn anzuhalten, weiterlaufen zu lassen und zu früheren Versionen zurückzurollen.

Fixed Deployment

Eine `RollingUpdate`-Strategie ist nützlich, um während des Update-Prozesses Zero Downtime sicherzustellen. Aber eine Nebenwirkung dieses Vorgehens ist, dass während des Update-Prozesses zwei Versionen der Container gleichzeitig laufen. Das kann für den Service-Konsumenten zu Problemen führen, insbesondere, wenn durch den Prozess inkompatible Änderungen an den Service-APIs vorgenommen wurden und der Client nicht damit umgehen kann. Für solche Szenarien gibt es die `Recreate`-Strategie, die in Abbildung 3-2 dargestellt ist.

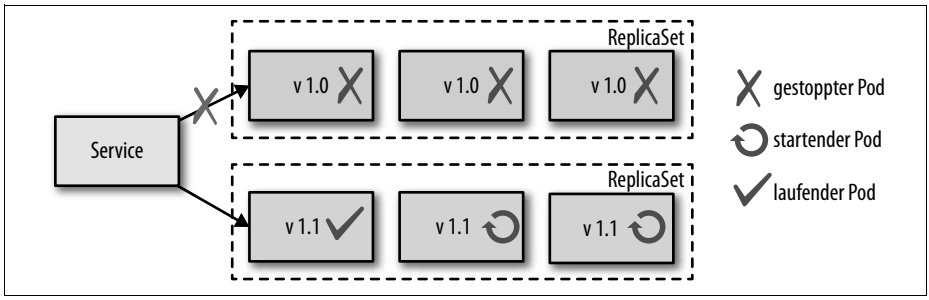

Abbildung 3-2: Fixed Deployment mit der Recreate-Strategie

Bei der Recreate-Strategie wird `maxUnavailable` auf die Anzahl der deklarierten Replicas gesetzt. Dadurch werden erst alle Container der aktuellen Version beendet und dann gleichzeitig alle neuen Container gestartet, wenn die alten Container nicht mehr vorhanden sind. Das Ergebnis dieser Abfolge ist, dass es eine gewisse Downtime gibt, in der alle Container der alten Version gestoppt wurden und noch keine neuen Container eintreffende Requests bearbeiten können. Der Vorteil ist aber, dass es keine zwei Versionen des Containers gibt, die gleichzeitig laufen, was das Leben der Service-Konsumenten deutlich vereinfacht, weil sie auch nur mit einer Version gleichzeitig rechnen müssen.

Blue/Green-Release

Das *Blue/Green-Deployment* ist eine Release-Strategie, die genutzt wird, um Software in einer Produktiv-Umgebung zu deployen, dabei die Downtime zu minimieren und das Risiko zu reduzieren. Die Deployment-Abstraktion von Kubernetes ist

ein so grundlegendes Konzept, dass Sie definieren können, wie Kubernetes immutable Container von einer Version zu einer anderen umwandeln soll. Sie können das Deployment-Primitiv zusammen mit anderen Primitiven von Kubernetes als Baustein nutzen, um diese ausgefeiltere Release-Strategie des Blue/Green-Deployments zu implementieren.

Ein Blue/Green-Deployment muss manuell durchgeführt werden, wenn keine Erweiterungen wie ein Service-Mesh oder Knative zum Einsatz kommen. Technisch gesehen erstellen Sie ein zweites Deployment mit der neuesten Version des Containers (nennen wir sie *Green*), die noch keine Requests bedient. In diesem Stadium laufen immer noch die alten Pod-Replicas (namens *Blue*) aus dem ursprünglichen Deployment und bedienen die Requests.

Sind Sie sicher, dass die neue Version der Pods läuft und dazu bereit ist, die Requests zu bedienen, leiten Sie den Traffic von den alten Pod-Replicas auf die neuen Replicas um. Das lässt sich in Kubernetes erreichen, indem Sie den Service-Selektor so anpassen, dass er zu den neuen Containern passt (untere Reihe). Wie in Abbildung 3-3 zu sehen, können die blauen Container (obere Reihe) gelöscht und deren Ressourcen für zukünftige Blue/Green-Deployments freigegeben werden, sobald die grünen Container (untere Reihe) den gesamten Traffic bedienen.

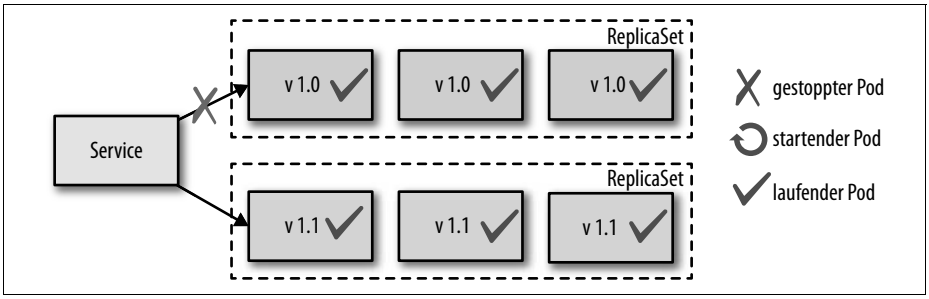

Abbildung 3-3: Blue/Green-Release

Ein Vorteil des Blue/Green-Vorgehens ist, dass es nur eine Version der Anwendung gibt, die Requests bedient, was die zusätzliche Komplexität durch mehrere konkurrierende Versionen für die Service-Konsumenten vermeidet. Nachteil ist, dass die doppelte Anwendungs-Kapazität verfügbar sein muss, während sowohl die blauen als auch die grünen Container laufen. Zudem kann es zu signifikanten Komplikationen mit langlaufenden Prozessen und Statusverschiebungen in der Datenbank während des Übergangs kommen.

Canary Release

Mit einem *Canary Release* wird eine neue Version einer Anwendung nach und nach produktiv geschaltet, indem nur eine kleine Untermenge der alten Instanzen durch neue ersetzt wird. Diese Technik verringert das Risiko beim Einführen einer neuen Version, indem nur ein paar der Konsumenten die aktualisierte Version

erhalten. Sind Sie mit der neuen Version Ihres Service und ihrer Performance bei den wenigen Nutzern zufrieden, ersetzen Sie alle alten Instanzen durch die neue Version. In Abbildung 3-4 sehen Sie ein Canary Release in Aktion.

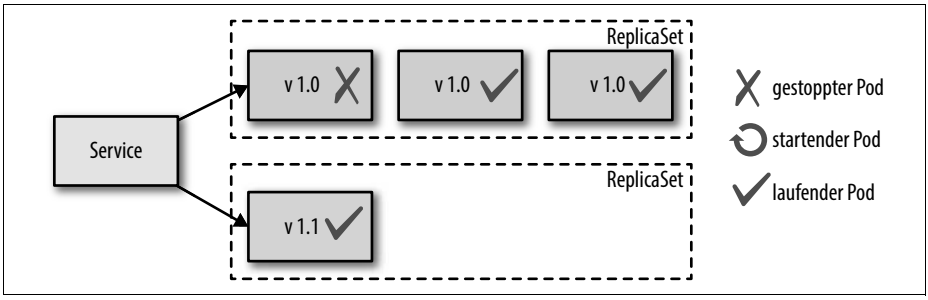

Abbildung 3-4: Canary Release

In Kubernetes kann diese Technik implementiert werden, indem man ein neues ReplicaSet für die neue Container-Version erstellt (möglichst über ein Deployment) und dabei nur eine kleine Replica-Anzahl angibt, die als Canary-Instanz genutzt werden kann. Dann sollte der Service einen Teil der Benutzer auf die aktualisierten Pod-Instanzen leiten. Sind Sie sicher, dass mit dem neuen ReplicaSet alles wie erwartet funktioniert, skalieren Sie das neue ReplicaSet hoch und das alte reduzieren wir auf null. So führen Sie ein kontrolliertes und durch Benutzer getestetes inkrementelles Rollout durch.

Diskussion

Das Deployment-Primitiv ist ein Beispiel dafür, wie Kubernetes den aufreibenden Prozess des manuellen Updatens von Anwendungen in eine deklarative Aktivität umwandelt, die wiederholt und automatisiert werden kann. Die mitgelieferten Deployment-Strategien (RollingUpdate und Recreate) steuern das Ersetzen alter Container durch neue, während die Release-Strategien (Blue/Green und Canary) steuern, wie die neuen Versionen für Service-Konsumenten verfügbar werden. Die letzten beiden Release-Strategien basieren auf einer Entscheidung für das Auslösen des Updates durch einen Menschen und sind daher nicht vollständig automatisiert. In Abbildung 3-5 sehen Sie eine Zusammenfassung der Deployment- und Release-Strategien mit den Instanz-Zählern während des Übergangs.

Jede Software ist unterschiedlich und das Deployen komplexer Systeme erfordert normalerweise zusätzliche Schritte und Prüfungen. Die in diesem Kapitel beschriebenen Techniken behandeln nur das Aktualisieren der Pods, aber nicht das Updaten oder den Rollback anderer Pod-Abhängigkeiten wie ConfigMaps, Secrets oder anderer abhängiger Services.

Aktuell gibt es ein Proposal für Kubernetes, Hooks im Deployment-Prozess zu erlauben. Pre- und Post-Hooks würden das Ausführen eigener Befehle ermöglichen, bevor und nachdem Kubernetes eine Deployment-Strategie durchgeführt hat. Da-

bei könnten zusätzliche Aktivitäten ausgeführt werden, während das Deployment läuft, und zusätzlich könnte man ein Deployment abbrechen, erneut laufen lassen oder fortführen. Diese Befehle sind ein guter Schritt hin zu neuen, automatisierten Deployment- und Release-Strategien. Aktuell ist die (machbare) Alternative, den Update-Prozess auf höherer Ebene mithilfe von Deployments und anderen Primitiven aus diesem Buch zu verskripten, um ihn für Services und seine Abhängigkeiten zu steuern.

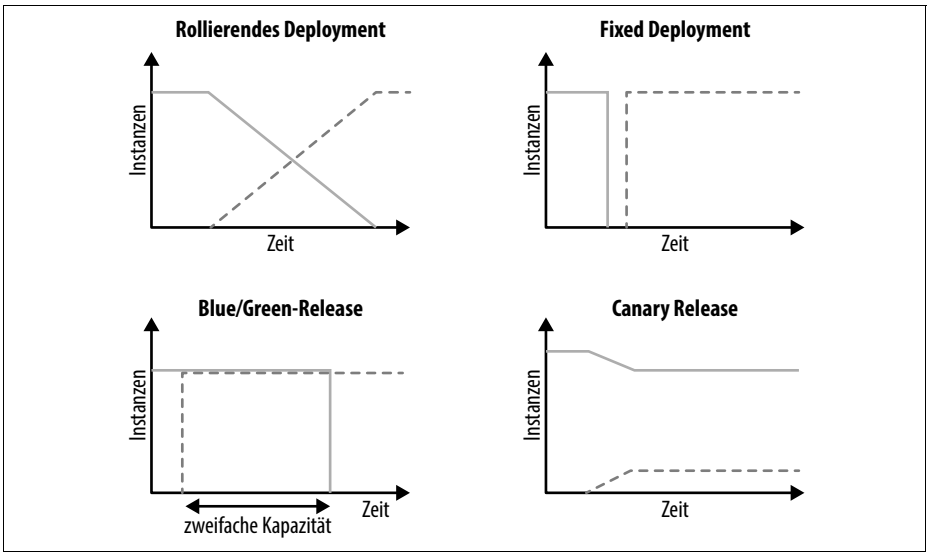

Abbildung 3-5: Deployment- und Release-Strategien

Unabhängig von der verwendeten Deployment-Strategie ist es für Kubernetes ausgesprochen wichtig, zu wissen, wann Ihre Anwendungs-Pods zur Verfügung stehen und laufen, um die erforderliche Abfolge von Schritten durchzuführen, die für das Erreichen des definierten Ziel-Zustands notwendig sind. Das nächste Pattern *Health Probe* in Kapitel 4 beschreibt, wie Ihre Anwendung diesen Health-Status an Kubernetes kommunizieren kann.

Weitere Informationen

- Beispiel für Declarative Deployment: *http://bit.ly/2Fc6d6J*
- Rolling Update: *http://bit.ly/2r06Ich*
- Deployments: *http://bit.ly/2q7vR7Y*
- Run a Stateless Application Using a Deployment: *http://bit.ly/2XZZhlL*
- Blue-Green Deployment: *http://bit.ly/1Gph4FZ*
- Canary Release: *https://martinfowler.com/bliki/CanaryRelease.html*
- DevOps with OpenShift: *https://red.ht/2W7fdAQ*