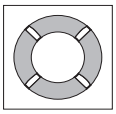
12 Deployments

Bisher haben Sie gesehen, wie Sie Ihre Anwendung als Container verpacken, Kopien dieses Containers schaffen und Services nutzen können, um den Verkehr per Load Balancing zu verteilen. Alle diese Objekte werden verwendet, um eine einzelne Instanz Ihrer Anwendung zu bauen. Sie helfen Ihnen nur wenig beim täglichen oder wöchentlichen Releasen neuer Versionen Ihrer Anwendung. Pods und ReplicaSets sind auch eigentlich dafür gedacht, mit bestimmten Container-Images verbunden zu sein, die sich nicht ändern.

Das `Deployment`-Objekt existiert, um das Releasen neuer Versionen zu managen. Deployments repräsentieren deployte Anwendungen über eine bestimmte Software-Version hinaus. Zudem ermöglichen es Deployments, einfach von einer Version Ihres Codes zur nächsten zu wechseln. Dieser »Rollout«-Prozess ist konfigurierbar und er geht vorsichtig vor. Er wartet eine definierbare Zeit zwischen dem Upgraden einzelner Pods ab. Zudem nutzt er Health-Checks, um sicherzustellen, dass die neue Version der Anwendung korrekt funktioniert, und er stoppt das Deployment, wenn zu viele Fehler auftreten.

Mit Deployments können Sie einfach und zuverlässig neue Software-Versionen ausrollen – ohne Downtime oder Fehler. Der eigentliche Rollout wird dabei von einem Deployment-Controller gesteuert, der selbst im Kubernetes-Cluster läuft. Das heißt, Sie können einen Deployment-Prozess unbeaufsichtigt laufen lassen, und er wird trotzdem korrekt und sicher arbeiten. Das erleichtert die Integration von Deployments in eine Reihe von Tools und Services zum Continuous Delivery. Zudem wird ein Rollout aufgrund der Steuerung über den Server auch dann sicher durchgeführt, wenn die Internetverbindung am eigenen Arbeitsplatz schwächelt oder zeitweilig nicht vorhanden ist. Stellen Sie sich vor, eine neue Version Ihrer Software von Ihrem Smartphone aus auszurollen, während Sie gerade in der U-Bahn sitzen. Mit Deployments wird das möglich und es ist dabei auch noch sicher!



Tipp

Als Kubernetes erstmals releast wurde, war eine der beliebtesten Demonstrationen seiner Leistungsfähigkeit das »rollierende Update«, bei dem gezeigt wurde, wie Sie mit einem einzigen Befehl eine laufende Anwendung reibungslos aktualisieren konnten – ohne Downtime oder das Verlieren von Requests. Diese ursprüngliche Demo basierte auf dem Befehl `kubectl rolling-update`, der immer noch zur Verfügung steht, dessen Funktionalität aber größtenteils in das `Deployment`-Objekt übergegangen ist.

12.1 Ihr erstes Deployment

Zu Beginn dieses Buches haben Sie einen Pod durch `kubectl run` erstellt. Der Befehl sah in etwas so aus:

```
$ kubectl run nginx --image=nginx:1.7.12
```

Hinter den Kulissen wurde dabei in Wirklichkeit ein `Deployment`-Objekt erstellt. Dieses Objekt können Sie sich wie folgt anzeigen lassen:

```
$ kubectl get deployments nginx
NAME   DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
Nginx  1        1        1           1          13s
```

12.1.1 Deployment-Interna

Schauen wir uns an, wie Deployments tatsächlich funktionieren. So wie wir gelernt haben, dass ReplicaSets Pods managen, managen Deployments ReplicaSets. Wie bei allen Beziehungen in Kubernetes wird diese Verbindung durch Labels und einen Label-Selektor definiert. Sie können sich diesen Selektor am `Deployment`-Objekt anschauen:

```
$ kubectl get deployments nginx \
  -o jsonpath --template {.spec.selector.matchLabels}

map[run:nginx]
```

Hieraus sehen Sie, dass das Deployment ein ReplicaSet mit dem Label `run=nginx` managt. Das können wir in einem Label-Selektor über alle ReplicaSets einsetzen, um das passende ReplicaSet zu finden:

```
$ kubectl get replicasets --selector=run=nginx

NAME              DESIRED  CURRENT  READY  AGE
nginx-1128242161  1        1        1      13m
```

Jetzt schauen wir uns die Beziehung zwischen einem Deployment und einem ReplicaSet in Aktion an. Wir können das Deployment über den imperativen Befehl `scale` in seiner Größe anpassen:

```
$ kubectl scale deployments nginx --replicas=2
deployment "nginx" scaled
```

Lassen wir uns das ReplicaSet erneut anzeigen, sollten wir Folgendes sehen:

```
$ kubectl get replicasets --selector=run=nginx

NAME              DESIRED  CURRENT  READY  AGE
nginx-1128242161  2        2        2      13m
```

Jetzt versuchen wir es in die Gegenrichtung:

```
$ kubectl scale replicasets nginx-1128242161 --replicas=1
replicaset "nginx-1128242161" scaled
```

Erneut ein Blick auf das ReplicaSet:

```
$ kubectl get replicasets --selector=run=nginx

NAME              DESIRED  CURRENT  READY  AGE
nginx-1128242161  2        2        2      13m
```

Seltsam. Obwohl wir das ReplicaSet auf eine Kopie verkleinert haben, gibt es immer noch zwei Replicas als gewünschten Status. Was passiert hier? Denken Sie daran: Kubernetes ist ein selbstheilendes Online-System. Das oberste `Deployment`-Objekt managt dieses ReplicaSet. Reduzieren Sie die Anzahl der Replicas auf eins, passt der gewünschte Status des Deployments nicht mehr, denn dort ist `replicas` auf `2` gesetzt. Der Deployment-Controller merkt das und stellt sicher, dass der beobachtete Status mit dem gewünschten Status übereinstimmt – in diesem Fall durch das erneute Anpassen der Anzahl an Replicas auf zwei.

Wollen Sie das ReplicaSet einmal direkt managen, müssen Sie das Deployment zuerst löschen (denken Sie daran, `--cascade` auf `false` zu setzen, ansonsten werden auch das ReplicaSet und die Pods gelöscht!).

12.2 Deployments erstellen

Wie schon erwähnt, sollten Sie natürlich ein deklaratives Management Ihrer Kubernetes-Konfigurationen bevorzugen. Das bedeutet, den Status Ihrer Deployments in YAML- oder JSON-Dateien vorzuhalten.

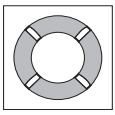
Als Ausgangspunkt laden Sie dieses Deployment in eine YAML-Datei herunter:

```
$ kubectl get deployments nginx --export -o yaml > \
    nginx-deployment.yaml
$ kubectl replace -f nginx-deployment.yaml --save-config
```

Schauen Sie sich die Datei an, werden Sie in etwa das Folgende sehen:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
```

```
    annotations:
      deployment.kubernetes.io/revision: "1"
    labels:
      run: nginx
    name: nginx
    namespace: default
  spec:
    replicas: 2
    selector:
      matchLabels:
        run: nginx
    strategy:
      rollingUpdate:
        maxSurge: 1
        maxUnavailable: 1
      type: RollingUpdate
    template:
      metadata:
        labels:
          run: nginx
      spec:
        containers:
        - image: nginx:1.7.12
          imagePullPolicy: Always
        dnsPolicy: ClusterFirst
        restartPolicy: Always
```



Tipp

Viele schreibgeschützte und Standard-Felder wurden aus Gründen der Lesbarkeit aus diesem Listing entfernt. Wir müssen zudem `kubectl replace --save-config` laufen lassen. Das fügt eine Anmerkung hinzu, durch die `kubectl` beim Anwenden von zukünftigen Änderungen weiß, welche die letzte angewandte Konfiguration war. So kann sie die Konfigurationen besser verschmelzen. Nutzen Sie immer `kubectl apply`, ist dieser Schritt nur das erste Mal erforderlich, wenn Sie ein Deployment über `kubectl create -f` erstellt haben.

Die Deployment-Spec hat eine sehr ähnliche Struktur wie eine ReplicaSet-Spec. Es gibt ein Pod-Template mit einer Reihe von Containern, die für jede durch das Deployment gemanagte Replica erstellt werden. Neben der Pod-Spezifikation gibt es noch das Objekt `strategy`:

```
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
```

Dieses Objekt gibt vor, wie der Rollout neuer Software ablaufen soll. Es gibt dabei zwei verschiedene Strategien, die von Deployments unterstützt werden: `Recreate` und `RollingUpdate`. Diese werden später noch in diesem Kapitel behandelt werden.

12.3 Deployments verwalten

Wie bei allen Kubernetes-Objekten können Sie detaillierte Informationen über Ihr Deployment über den Befehl `kubectl describe` erhalten:

```
$ kubectl describe deployments nginx

Name:                   nginx
Namespace:              default
CreationTimestamp:      Sat, 31 Dec 2016 09:53:32 -0800
Labels:                 run=nginx
Selector:               run=nginx
Replicas:               2 updated | 2 total |
                        2 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
OldReplicaSets:         <none>
NewReplicaSet:          nginx-1128242161 (2/2 replicas created)
Events:
  FirstSeen   ...   Message
  ---------   ...   -------
  5m          ...   Scaled up replica set nginx-1128242161 to 1
  4m          ...   Scaled up replica set nginx-1128242161 to 2
```

Die Ausgabe von `describe` bietet eine Menge Informationen. Die beiden wichtigsten Elemente sind `OldReplicaSets` und `NewReplicaSets`. Diese Felder verweisen auf die ReplicaSet-Objekte, die dieses Deployment aktuell verwaltet. Befindet sich ein Deployment mitten in einem Rollout, werden beide Felder einen Wert enthalten. Nach Abschluss des Rollouts wird `OldReplicaSets` auf `<none>` gesetzt.

Neben dem `describe`-Befehl gibt es auch noch den Befehl `kubectl rollout` für Deployments. Wir werden ihn uns später genauer anschauen, hier wollen wir darauf hinweisen, dass Sie mit `kubectl rollout history` den Verlauf von Rollouts anzeigen lassen können, die mit einem bestimmten Deployment verbunden sind. Befindet sich ein Deployment gerade in der Durchführung eines Rollouts, können Sie mit `kubectl rollout status` den aktuellen Status abfragen.

12.4 Deployments aktualisieren

Bei Deployments handelt es sich um deklarative Objekte, die eine deployte Anwendung beschreiben. Die beiden wichtigsten Operationen bei einem Deployment-Objekt sind das Skalieren und das Anwenden von Updates.

12.4.1 Ein Deployment skalieren

Wir haben schon gezeigt, wie Sie ein Deployment mit dem Befehl `kubectl scale` imperativ skalieren können, aber besser ist es, Ihre Deployments deklarativ über die YAML-Dateien zu managen und diese dann zum Aktualisieren Ihres Deployments einzusetzen. Um ein Deployment nach oben zu skalieren, bearbeiten Sie Ihre YAML-Datei und erhöhen den Wert der Replicas:

```
...
spec:
  replicas: 3
...
```

Nach dem Speichern und Einchecken Ihrer Änderung können Sie das Deployment mithilfe des Befehls `kubectl apply` aktualisieren:

```
$ kubectl apply -f nginx-deployment.yaml
```

Damit wird der gewünschte Status des Deployments angepasst, was dazu führt, dass die Größe des verwalteten ReplicaSets erhöht und schließlich ein neuer, durch das Deployment gemanagte Pod erzeugt wird:

```
$ kubectl get deployments nginx

NAME   DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
Nginx  3        3        3           3          4m
```

12.4.2 Ein Container-Image aktualisieren

Der andere häufige Anwendungsfall für das Aktualisieren eines Deployments ist das Ausrollen einer neuen Version der Software, die in einem oder mehreren Containern läuft. Dazu sollten Sie auch die Deployment-YAML-Datei bearbeiten, dieses Mal aber das Container-Image anpassen:

```
...
  containers:
  - image: nginx:1.9.10
    imagePullPolicy: Always
...
```

Wir werden noch eine Anmerkung in das Template für das Deployment einfügen, um Informationen über das Update einzutragen:

```
...
spec:
  ...
  template:
    annotations:
      kubernetes.io/change-cause: "nginx auf 1.9.10 aktualisiert"
...
```

Warnung

Achten Sie darauf, diese Anmerkung in das Template einzutragen, nicht in das Deployment selbst. Aktualisieren Sie die `change-clause` aber nicht bei einer einfachen Skalierungsaktion. Eine Veränderung von `change-clause` ist eine für das Template signifikante Änderung, die einen neuen Rollout auslöst.

Wieder können Sie mit `kubectl apply` das Deployment aktualisieren:

```
$ kubectl apply -f nginx-deployment.yaml
```

Nach dem Aktualisieren des Deployments wird ein Rollout ausgelöst, den Sie über den Befehl `kubectl rollout` überwachen können:

```
$ kubectl rollout status deployments nginx
deployment nginx successfully rolled out
```

Sie können die alten und neuen durch das Deployment verwalteten ReplicaSets und die eingesetzten Images sehen. Sowohl das alte wie auch das neue ReplicaSet wird aufgehoben, falls Sie den Status zurückrollen wollen:

```
$ kubectl get replicasets -o wide

NAME              DESIRED  CURRENT  READY  …  IMAGE(S)      …
nginx-1128242161  0        0        0      …  nginx:1.7.12  …
nginx-1128635377  3        3        3      …  nginx:1.9.10  …
```

Befinden Sie sich mitten in einem Rollout und wollen diesen aus irgendeinem Grund temporär pausieren lassen (zum Beispiel, weil Sie sehen, dass sich Ihr System seltsam verhält, und Sie sich das genauer anschauen wollen), können Sie den `pause`-Befehl nutzen:

```
$ kubectl rollout pause deployments nginx
deployment "nginx" paused
```

Wenn Sie nach der Untersuchung der Meinung sind, dass der Rollout sicher weiterlaufen kann, nutzen Sie `resume`, um dort weiterzumachen, wo Sie aufgehört haben:

```
$ kubectl rollout resume deployments nginx
deployment "nginx" resumed
```

12.4.3 Rollout-History

Die Deployments in Kubernetes merken sich eine Historie ihrer Rollouts, was nützlich sein kann, um den früheren Status des Deployments zu verstehen, und um auf eine bestimmte Version zurückzurollen.

Die Deployment-Historie sehen Sie über:

```
$ kubectl rollout history deployment nginx

deployments "nginx"
REVISION  CHANGE-CAUSE
1         <none>
2         nignx auf 1.9.10 aktualisiert
```

Die Revisionshistorie wird von der ältesten zur neuesten Version ausgegeben. Für jeden neuen Rollout wird eine eindeutige Revisionsnummer erhöht. Bisher haben wir zwei: das initiale Deployment und das Update des Image auf `nginx:1.9.10`.

Sind Sie an mehr Details zu einer bestimmten Revision interessiert, können Sie das Flag `--revision` mit der gewünschten Nummer nutzen:

```
$ kubectl rollout history deployment nginx --revision=2

deployments "nginx" with revision #2
  Labels:       pod-template-hash=2738859366
        run=nginx
  Annotations:  kubernetes.io/change-cause=nginx auf 1.9.10 aktua…
  Containers:
    nginx:
      Image:      nginx:1.9.10
      Port:
      Volume Mounts:      <none>
      Environment Variables:    <none>
No volumes.
```

Führen wir noch ein weiteres Update für dieses Beispiel durch. Erhöhen wir die Version von nginx auf 1.10.2, indem wir die Versionsnummer des Containers und die Anmerkung `change-cause` anpassen. Wenden Sie das Ganze mit `kubectl apply` an. Unsere Historie sollte nun drei Einträge enthalten:

```
$ kubectl rollout history deployment nginx

deployments "nginx"
REVISION  CHANGE-CAUSE
1         <none>
2         nginx auf 1.9.10 aktualisiert
3         nginx auf 1.10.2 aktualisiert
```

Nehmen wir an, es gibt ein Problem mit dem letzten Release und Sie wollen die Version zurückrollen, um genauere Nachforschungen anzustellen. Sie können ganz einfach den letzten Rollout rückgängig machen:

```
$ kubectl rollout undo deployments nginx
deployment "nginx" rolled back
```

Der undo-Befehl funktioniert unabhängig vom Status des Rollouts. Sie können sowohl nur teilweise abgeschlossene Rollouts wie auch vollständige Rollouts zurückrollen. Bei einem Undo eines Rollouts handelt es sich tatsächlich einfach nur um einen umgekehrten Rollout (zum Beispiel von *v2* nach *v1* statt von *v1* nach *v2*) und alle Regeln, die die Rollout-Strategie steuern, gelten auch für die Undo-Strategie. Sie können sehen, wie das `Deployment`-Objekt einfach die Anzahl der gewünschten Replicas in den verwalteten ReplicaSets anpasst:

```
$ kubectl get replicasets -o wide

NAME              DESIRED  CURRENT  READY  …  IMAGE(S)      …
nginx-1128242161  0        0        0      …  nginx:1.7.12  …
nginx-1570155864  0        0        0      …  nginx:1.10.2  …
nginx-2738859366  3        3        3      …  nginx:1.9.10  …
```

Warnung

Nutzen Sie deklarative Dateien, um Ihre Produktiv-Systeme zu steuern, sollten Sie darauf achten, dass die eingecheckten Manifeste dem entsprechen, was gerade in Ihrem Cluster läuft. Führen Sie ein `kubectl rollout undo` durch, aktualisieren Sie den Produktiv-Status auf eine Art und Weise, die sich nicht in Ihrer Versionsverwaltung widerspiegelt.

Eine alternative (und eventuell zu bevorzugende) Variante für das Rückgängigmachen eines Rollouts ist, Ihre YAML-Datei auf einen älteren Stand zurückzusetzen und diesen mit `kubectl apply` anzuwenden. So passt Ihr Stand in der Versionsverwaltung besser zu dem, was aktuell im Cluster passiert.

Schauen wir uns unsere Deployment-Historie erneut an:

```
$ kubectl rollout history deployment nginx

deployments "nginx"
REVISION  CHANGE-CAUSE
1         <none>
3         nginx auf 1.10.2 aktualisiert
4         nginx auf 1.9.10 aktualisiert
```

Revision 2 fehlt! Es stellt sich heraus, dass das Deployment beim Zurückrollen auf einen älteren Stand einfach das Template wiederverwendet und mit einer neuen Nummer versieht, sodass es zur neuesten Version wird. Was zuvor Revision 2 war, wurde nun zu Revision 4.

Wir haben schon gesehen, dass Sie den Befehl `kubectl rollout undo` nutzen können, um zur letzten Version eines Deployments zurückzurollen. Zusätzlich können Sie zu einer bestimmten Revision in der Historie zurückgehen, indem Sie das Flag `--to-revision` nutzen:

```
$ kubectl rollout undo deployments nginx --to-revision=3
deployment "nginx" rolled back
$ kubectl rollout history deployment nginx
deployments "nginx"
REVISION  CHANGE-CAUSE
1         <none>
4         nginx auf 1.9.10 aktualisiert
5         nginx auf 1.10.2 aktualisiert
```

Wieder hat sich das Undo Revision 3 genommen, es angewendet und als neue Revision 5 notiert.

Geben Sie als Revisionsnummer die `0` an, gelangen Sie immer zur vorigen Version. Daher entspricht `kubectl rollout undo` dem Befehl `kubectl rollout undo --to-revision=0`.

Standardmäßig wird die gesamte Revisionshistorie eines Deployments am Deployment-Objekt selbst verwaltet. Mit der Zeit (zum Beispiel über mehrere Jahre) kann diese Historie ziemlich groß werden, daher empfiehlt es sich, bei langlebigen Deployments eine maximale Größe für die Revisionshistorie festzulegen, um die Gesamtgröße des `Deployment`-Objekts im Griff zu behalten. Führen Sie zum Beispiel ein tägliches Update durch, begrenzen Sie die Revisionshistorie vielleicht auf 14, um die Revisionen von zwei Wochen vorzuhalten (wenn Sie nicht davon ausgehen, dass Sie weiter als zwei Wochen in die Vergangenheit zurückrollen müssen). Um das zu erreichen, nutzen Sie die Eigenschaft `revisionHistoryLimit` in der Deployment-Spezifikation:

```
...
spec:
  # Wir führen tägliche Rollouts durch, beschränken die Revisions-
  # historie auf zwei Wochen, da wir nicht davon ausgehen, weiter
  # zurückgehen zu müssen
  revisionHistoryLimit: 14
...
```

12.5 Deployment-Strategien

Wenn es daran geht, die Version der Software zu ändern, die Ihren Service implementiert, unterstützt ein Kubernetes-Deployment zwei verschiedene Rollout-Strategien:

- `Recreate`
- `RollingUpdate`

12.5.1 Recreate-Strategie

Die `Recreate`-Strategie ist die einfachere der beiden Strategien. Sie aktualisiert das durch das Deployment gemanagte ReplicaSet so, dass das neue Image verwendet wird, und beendet alle Pods, die mit dem Deployment verbunden sind. Das ReplicaSet bemerkt, dass es keine Replicas mehr besitzt, und erstellt alle Pods mit dem neuen Image. Sind die Pods wieder erzeugt worden, laufen sie mit der neuen Version.

Diese Strategie ist zwar einfach und schnell, hat aber einen entscheidenden Nachteil – sie kann potenziell katastrophale Folgen haben und wird mit ziemlicher Sicherheit zu etwas Downtime führen. Daher sollte sie nur für Test-Deployments genutzt werden, bei denen ein Service nicht von Anwendern genutzt wird und eine geringe Downtime akzeptabel ist.

12.5.2 RollingUpdate-Strategie

Die `RollingUpdate`-Strategie ist zu bevorzugen, wenn ein Service von Endanwendern genutzt wird. Sie ist zwar langsamer als `Recreate`, dafür aber deutlich ausgefeilter und robuster. Mit `RollingUpdate` können Sie eine neue Version Ihres Service ausrollen, während dieser weiterhin Benutzer-Traffic empfängt, womit Downtime vermieden wird. Wie der Name besagt, funktioniert diese Strategie, indem Kubernetes immer nur ein paar Pods gleichzeitig aktualisiert, sodass nach und nach alle Pods auf die neue Software-Version umgestellt werden.

Mehrere Versionen Ihres Service managen

Beachten Sie, dass so für einen gewissen Zeitraum gleichzeitig die neue und die alte Version Ihres Service Requests empfangen und Antworten liefern. Das hat entscheidende Auswirkungen darauf, wie Sie Ihre Software bauen sollten. Insbesondere ist es außerordentlich wichtig, dass jede Version Ihrer Software und alle beteiligten Clients mit etwas älteren und etwas neueren Versionen Ihrer Software umgehen können. Um zu zeigen, warum das so wichtig ist, stellen Sie sich folgendes Szenario vor:

> *Sie befinden sich mitten im Rollout Ihrer Frontend-Software und auf der einen Hälfte Ihrer Server läuft noch Version 1, während auf der anderen Hälfte schon Version 2 läuft. Ein Benutzer schickt einen ersten Request an Ihren Service und lädt eine clientseitige JavaScript-Bibliothek herunter, die Ihr UI implementiert. Dieser Request wird durch einen Server mit Version 1 bedient, daher erhält der Anwender die Client-Bibliothek in Version 1. Diese läuft im Browser des Benutzers und macht API-Aufrufe gegen Ihren Service, die nun durchaus auch an einen Server in Version 2 geraten können – die Version 1 Ihrer JavaScript-Client-Bibliothek redet nun mit Version 2 Ihres API-Servers. Haben Sie keine Kompatibilität zwischen diesen Versionen sichergestellt, wird Ihre Anwendung nicht korrekt funktionieren.*

Zunächst scheint das wie eine zusätzliche Last auszusehen. Aber in Wahrheit hatten Sie dieses Problem schon immer – es ist Ihnen vielleicht nur noch nicht aufgefallen. Ein Beispiel: Der Anwender setzt einen Request zur Zeit *t* ab – kurz vor dem Update. Dieser Request wird von einem Server mit Version 1 bedient. Bei *t*1 aktualisieren Sie Ihren Service auf Version 2. Bei *t*2 wird der Client-Code (in Version 1) im Browser des Clients wieder ausgeführt und jetzt erreicht er einen API-Endpunkt, der durch einen Server mit Version 2 bedient wird. Egal, wie Sie Ihre Software updaten – Sie müssen für zuverlässige Updates eine gewisse Rückwärts- und Vorwärts-Kompatibilität bieten. Die Natur der rollierenden Update-Strategie macht nur deutlicher, dass es da etwas gibt, worüber Sie nachdenken müssen.

Beachten Sie, dass das nicht nur für JavaScript-Clients gilt – auch bei Client-Bibliotheken, die in andere Services kompiliert wurden und Ihren Service aufrufen, kann das passieren. Nur weil Sie Ihren Service aktualisiert haben, bedeutet dies nicht, dass auch die Client-Bibliotheken erneuert wurden. Diese Form der Rückwärts-Kompatibilität ist entscheidend, wenn Sie Ihren Service von den Systemen entkoppeln, die ihn verwenden. Formalisieren Sie Ihre APIs nicht und entkoppeln selbst, sind Sie dazu gezwungen, Ihre Rollouts sorgfältig mit allen anderen Systemen abzustimmen, die Ihren Service aufrufen. Diese Art enger Kopplung macht es ausgesprochen schwer, für die notwendige Agilität zu sorgen, mit der Sie jede Woche neue Software ausliefern können – von einem täglichen oder stündlichen Deployment gar nicht erst zu reden. In der entkoppelten Architektur in Abbildung 12–1 ist das Frontend vom Backend über einen API-Kontrakt und einen Load Balancer isoliert, während sich in der gekoppelten Architektur ein Thick Client, der in das Frontend kompiliert ist, direkt mit den Backends verbindet.

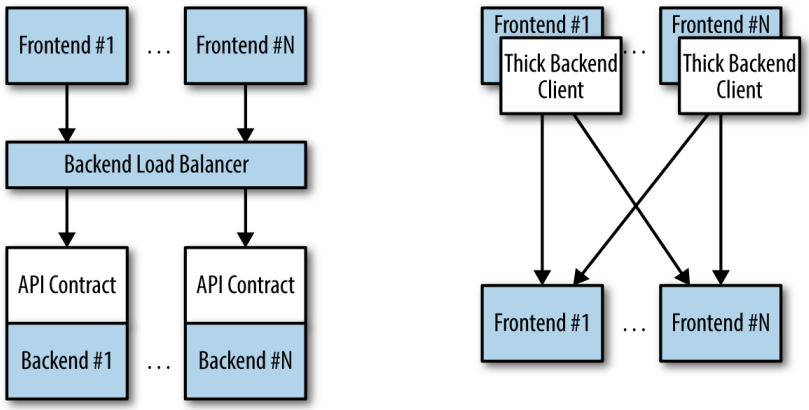


Abb. 12–1 *Entkoppelte (links) und gekoppelte (rechts) Anwendungsarchitektur*

Ein rollierendes Update konfigurieren

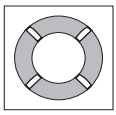
`RollingUpdate` ist eine ziemlich generische Strategie – sie kann genutzt werden, um verschiedenste Anwendungen in unterschiedlichsten Situationen zu aktualisieren. Daher ist das rollierende Update ziemlich umfassend konfigurierbar – Sie können sein Verhalten auf Ihre speziellen Bedürfnisse anpassen. Es gibt zwei Parameter, mit denen Sie sehr viele Varianten abdecken: `maxUnavailable` und `maxSurge`.

Der Parameter `maxUnavailable` definiert die maximale Anzahl an Pods, die während eines rollierenden Updates nicht verfügbar sein dürfen. Dabei kann es sich entweder um eine absolute Zahl (zum Beispiel bedeutet `3`, dass höchstens drei Pods nicht verfügbar sein dürfen) oder um einen Prozentwert handeln (mit `20%` stehen höchstens 20 % der gewünschten Anzahl an Replicas nicht zur Verfügung).

Meist ist die Angabe einer Prozentzahl ein guter Ansatz, da der Wert unabhängig von der gewünschten Anzahl an Replicas im Deployment anwendbar ist. Aber es gibt Situationen, in denen Sie eine absolute Zahl nutzen wollen (zum Beispiel, weil höchstens ein Pod gleichzeitig nicht verfügbar sein darf).

Mit dem Parameter `maxUnavailable` bestimmen Sie auch, wie schnell ein rollierendes Update vorankommt. Setzen Sie zum Beispiel `maxUnavailable` auf `50%`, wird das rollierende Update das alte ReplicaSet direkt auf 50 % verkleinern. Haben Sie vier Replicas, werden nur zwei übrig bleiben. Dann ersetzt das Update die entfernten Pods, indem es das neue ReplicaSet auf zwei Replicas hochskaliert, damit insgesamt vier Replicas zur Verfügung stehen (zwei alte, zwei neue). Dann wird das alte ReplicaSet auf 0 Replicas heruntergefahren und es bleiben nur die beiden neuen Replicas. Schließlich wird das neue ReplicaSet auf vier Replicas hochgesetzt und der Rollout damit abgeschlossen. Durch Setzen von `maxUnavailable` auf `50%` ist unser Rollout also in vier Schritten abgeschlossen, aber die Servicekapazität ist in der Zeit auf 50 % reduziert.

Was passiert hingegen, wenn wir `maxUnavailable` auf `25%` setzen? In dieser Situation wird jeder Schritt immer nur mit einer Replica gleichzeitig durchgeführt und es werden doppelt so viele Schritte benötigt, um den Rollout abzuschließen. Dafür ist die Verfügbarkeit aber nur auf 75 % reduziert. Dies zeigt, wie es `maxUnavailable` erlaubt, die Rollout-Geschwindigkeit mit der Verfügbarkeit abzustimmen.



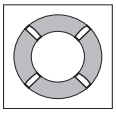
Tipp

Vielleicht ist Ihnen aufgefallen, dass die Recreate-Strategie identisch ist zur rollierenden Update-Strategie, bei der `maxUnavailable` auf `100%` gesetzt ist.

Das Akzeptieren einer verringerten Kapazität für einen erfolgreichen Rollout ist nützlich, wenn Ihr Service entweder zyklische Traffic-Muster hat (zum Beispiel viel weniger Auslastung zur Nacht) oder wenn Sie nur begrenzte Ressourcen besitzen, sodass es nicht möglich ist, mehr als die aktuell maximale Anzahl an Replicas zu nutzen.

Aber es gibt Situationen, in denen Sie nicht unter 100 % Kapazität fallen wollen, aber dazu bereit sind, temporär zusätzliche Ressourcen einzusetzen, um einen Rollout durchzuführen. Dann können Sie `maxUnavailable` auf `0%` setzen und stattdessen den Rollout über den Parameter `maxSurge` steuern. Wie `maxUnavailable` können Sie für `maxSurge` entweder eine absolute Zahl oder einen Prozentwert angeben.

`maxSurge` legt fest, wie viele zusätzliche Ressourcen erstellt werden können, um einen Rollout durchzuführen. Um zu zeigen, wie das funktioniert, stellen Sie sich vor, dass wir einen Service mit zehn Replicas haben. Wir setzen `maxUnavailable` auf `0` und `maxSurge` auf `20%`. Als Erstes wird der Rollout nun das neue ReplicaSet auf zwei Replicas setzen, sodass nun 12 (120 %) Replicas für den Service im Einsatz sind. Dann wird das alte ReplicaSet auf acht Replicas heruntergefahren, sodass insgesamt zehn zur Verfügung stehen (acht alte, zwei neue). Dieser Prozess wird fortgesetzt, bis der Rollout abgeschlossen ist. Die Kapazität des Service liegt jederzeit bei mindestens 100 % und die zusätzlichen Ressourcen für den Rollout bei maximal 20 % aller Ressourcen.



Tipp

Setzen Sie `maxSurge` auf `100%`, haben Sie ein Blue/Green-Deployment. Der Deployment-Controller skaliert zuerst die neue Version auf 100 %. Läuft sie erfolgreich, wird die alte Version direkt auf 0 % heruntergefahren.

12.5.3 Rollouts verlangsamen, um die Service-Qualität sicherzustellen

Der Zweck eines abgestuften Rollouts liegt darin, sicherzustellen, dass er zu einem gesunden und stabilen Service führt, der die neue Software-Version nutzt. Dazu wartet der Deployment-Controller immer, bis ein Pod seine Arbeitsbereitschaft meldet, bevor er mit dem Update des nächsten Pods fortfährt.

Warnung

Der Deployment-Controller wertet den Status des Pods über dessen Readiness-Checks aus. Readiness-Checks sind Teil der Health-Proben eines Pods und detaillierter in Kapitel 5 beschrieben. Wollen Sie Deployments nutzen, um Ihre Software zuverlässig auszurollen, *müssen* Sie Readiness-Checks für die Container in Ihrem Pod definieren. Ohne diese Checks geht der Deployment-Controller blind vor.

Manchmal reicht es aber nicht, einfach nur zu warten, bis ein Pod arbeitsbereit ist, weil Sie nicht sicher sein können, ob er sich auch wirklich korrekt verhält. Manche Fehlerbedingungen treten erst nach einer gewissen Zeit auf. So könnten Sie zum Beispiel ein ernsthaftes Speicherleck haben, das erst nach ein paar Minuten zutage tritt, oder Sie haben einen Bug, der nur bei 1 % der Requests zu bemerken ist. In den meisten echten Update-Szenarien werden Sie eine gewisse Zeit warten wollen, um sicher zu sein, dass die neue Version korrekt arbeitet, bevor Sie den nächsten Pod aktualisieren.

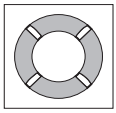
Für Deployments wird diese Wartezeit durch den Parameter `minReadySeconds` definiert:

```
...
spec:
  minReadySeconds: 60
...
```

Mit diesem Wert wird angegeben, dass das Deployment noch 60 Sekunden warten muss, *nachdem* ein Pod sich erfolgreich zurückgemeldet hat, bevor es den nächsten Pod angeht.

Neben der Definition einer Wartezeit wollen Sie vielleicht auch einen Timeout festlegen, der begrenzt, wie lange das System warten wird. Stellen Sie sich zum Beispiel vor, dass die neue Version Ihres Service einen Bug hat, der unmittelbar zu einem Deadlock führt. Der Pod wird so niemals seine Arbeitsbereitschaft melden können und ohne Timeout wird der Deployment-Controller Ihren Rollout für immer warten lassen.

Das korrekte Verhalten ist in einem solchen Fall, einen Timeout zuschlagen zu lassen. Das wiederum kennzeichnet den Rollout als fehlgeschlagen. Dieser Fehlerstatus kann dann genutzt werden, um eine Warnung an einen Administrator zu verschicken.



Tipp

Auf den ersten Blick scheint es einen Rollout unnötig zu verkomplizieren, auch noch ein Timeout zu setzen. Aber solche Dinge wie Rollouts werden zunehmend durch automatisierte Systeme ausgelöst, bei denen Menschen nur wenig beteiligt sind. In solch einer Situation ist ein Timeout wichtig, denn so kann entweder ein automatisches Rollback ausgelöst oder ein Ticket/Event erzeugt werden, damit sich ein Mensch die Sache mal ansieht.

Um die Timeout-Periode zu setzen, wird der Deployment-Parameter `progressDeadlineSeconds` verwendet:

```
...
spec:
  progressDeadlineSeconds: 600
...
```

Dieses Beispiel setzt die Deadline auf zehn Minuten. Wenn eine Stufe des Rollouts mehr als zehn Minuten ohne Fortschritt ist, wird das Deployment als fehlgeschlagen markiert und alle weiteren geplanten Schritte werden abgebrochen.

Achten Sie darauf, dass sich dieser Timeout auf den Deployment-*Fortschritt* bezieht, nicht auf die gesamte Dauer des Deployments. Dabei ist Fortschritt definiert als die Momente, in denen ein Pod erstellt oder zerstört wird. Wenn das passiert, wird der Timeout-Zähler zurückgesetzt. Abbildung 12–2 zeigt, wie der Deployment-Lebenszyklus aussieht.

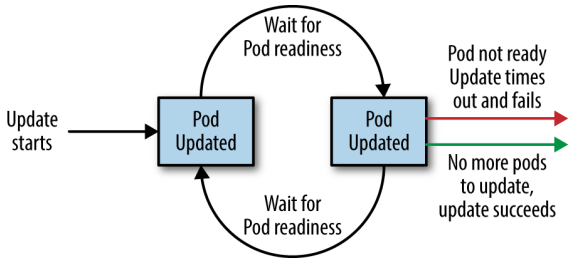


Abb. 12–2 *Der Deployment-Lebenszyklus von Kubernetes*

12.6 Ein Deployment löschen

Wenn Sie ein Deployment löschen wollen, können Sie das entweder über einen imperativen Befehl erreichen:

```
$ kubectl delete deployments nginx
```

Oder Sie verwenden die deklarative YAML-Datei, die wir weiter oben erstellt haben:

```
$ kubectl delete -f nginx-deployment.yaml
```

In beiden Fällen werden standardmäßig neben dem Deployment selbst auch der gesamte Service und damit alle durch das Deployment gemanagten ReplicaSets und die dadurch verwalteten Pods gelöscht. Wie bei ReplicaSets gilt auch hier: Ist das nicht das gewünschte Verhalten, können Sie das Flag `--cascade=false` nutzen, um nur das Deployment-Objekt selbst zu löschen.

12.7 Zusammenfassung

Das eigentliche Ziel von Kubernetes ist, das Bauen und Deployen zuverlässiger, verteilter Systeme für Sie einfach zu machen. Dazu gehört nicht nur das einmalige Instanziieren der Anwendung, sondern auch das Managen regelmäßig geschedulter Rollouts neuer Versionen. Deployments sind für zuverlässige Rollouts und für das Rollout-Management für Ihre Services ein zentraler Bestandteil.