GitOps mit Helm und Kubernetes

Bernd Stübinger und Florian Heubeck, Java User Group Ingolstadt e.V.

DevOps, also das Entwickeln und Betreiben von Software als Teamaufgabe, verlangt entweder nach Spezialisten aus beiden Bereichen oder interdisziplinären Experten, die beides abdecken. Da dies in der Realität schwer zu erreichen ist, erfordert echtes DevOps Werkzeuge, die einen weitestgehend automatischen Betrieb ermöglichen und die Laufzeitumgebung einem Entwicklungsteam zugänglich machen. Hier hilft das Konzept GitOps, dessen Einsatz wir auf Basis von Helm und Kubernetes erläutern wollen.

DevOps – ein Begriff, der sich inzwischen bis in die letzten Winkel des IT-Managements verbreitet hat. Doch was bedeutet DevOps für uns Entwickler? Entwicklung und Betrieb aus einer Hand! Um aktuellen Trends zu folgen, läuft Software heutzutage natürlich in der Cloud, und somit landet man schlussendlich bei einem Team, das eine Menge unterschiedlicher Dinge in einen Kubernetes-Cluster schiebt und auf das Beste hofft.

Wer es ein wenig organisierter mag, fängt an, seine Software in Helm-Charts zu bündeln. Damit hat man dann zumindest eine Menge zusammengehöriger Dinge, die versioniert und einfacher überblickt werden können. Helm allein ermöglicht allerdings auch noch kein kontinuierliches und zuverlässiges Ausspielen konkreter Softwarestände. Was tun, wenn nur bestimmte Ressourcen erfolgreich

aktualisiert werden konnten? Was, wenn sich die Konfiguration ändert? Und wie stellt man überhaupt fest, welche Version und Konfiguration aktuell läuft? Was typischerweise auch weniger gut funktioniert, ist das Wiederaufsetzen nach einem größeren Ausfall. Oder: Wie lange dauert es, bis ein leerer Cluster wieder produktiv wird?

CI != CD

Aus der klassischen Softwareentwicklung sind wir es gewohnt, Build-Pipelines zu instrumentalisieren. Diese lösen schließlich bereits das Problem der Continuous Integration. Integriert man dort auch noch Delivery und Deployment, besteht die Gefahr, dass diese Pipeline aufgrund diverser Anforderungen und verschiedener Unwägbarkeiten schnell eine Komplexität erreicht, die weder überschau- noch wartbar ist.

Das Bauen eines Artefaktes ist ein geradliniger Ablauf, mit zwei möglichen Ergebnissen: einer neuen Version des zu bauenden Artefaktes oder eben einem Fehlschlag. Viele Anbieter stellen Build Services zur Verfügung, die unkompliziert an das eigene Git Repository angebunden werden können und diese Aufgabe zuverlässig erledigen.

Die Komplexität von Continuous Delivery ist jedoch eine ganz andere als die des Builds. Fehler müssen behandelt, Zustände gesichert und gegebenenfalls wiederhergestellt werden. Hinzu kommt, dass sich Deployments womöglich je nach Stage unterschiedlich verhalten müssen. Die Entwicklungsumgebung kann anders behandelt werden als die Produktion, und jede weitere Ausprägung ist denkbar. Zusätzlich wollen noch eventuelle Zugangsdaten hinterlegt werden, und man will oft lieber nicht so genau darüber nachdenken, was eigentlich zu tun wäre, falls sich einmal der komplette Cluster spontan verabschiedet.

```
helm repo add fluxcd https://fluxcd.github.io/flux
helm upgrade -i flux --namespace flux \
  --set helmOperator.create=true \
  --set helmOperator.createCRD=true \
  --set syncGarbageCollection.enabled=true \
  --set git.url=<GitOps Repo URI> \
  --set git.branch=master fluxcd/flux
```

Listing 1: Installation von Flux via Helm

Kommen wir zum Punkt: Um einen vernünftigen Betrieb gewährleisten zu können, bedarf es einer zuverlässigen Lieferkette und einfacher Konfigurationsmöglichkeiten aus Sicht des Entwicklungsteams.

Die Lösung für diese Problemstellung heißt GitOps. Ein Begriff, der zuerst von Weaveworks geprägt wurde *[1]* und im Prinzip die konsequente Fortführung des Konzepts von „Infrastructure as Code" darstellt, das sich besonders im Cloud-Umfeld etabliert hat: Der Entwickler beschreibt den gewünschten Zustand der Infrastruktur deklarativ und spezialisierte Software, wie zum Beispiel Ansible oder Terraform, kümmert sich um die zuverlässige Bereitstellung. Warum sollte dieser Ansatz nicht auch für die auf dieser Infrastruktur laufende Software möglich sein – insbesondere wenn Kubernetes ohnehin bereits alle Ressourcen deklarativ beschreibt.

Mit GitOps werden neben dem Sourcecode nun auch die Laufzeitkonfiguration und der gewünschte Zustand des Clusters in Git hinterlegt und damit transparent, automatisch versioniert und vor allem nachvollziehbar. Nachvollziehbar sowohl für Menschen als auch für Tools, die darauf aufbauend automatisch den Soll- mit dem Ist-Zustand abgleichen und gegebenenfalls korrigierend eingreifen können.

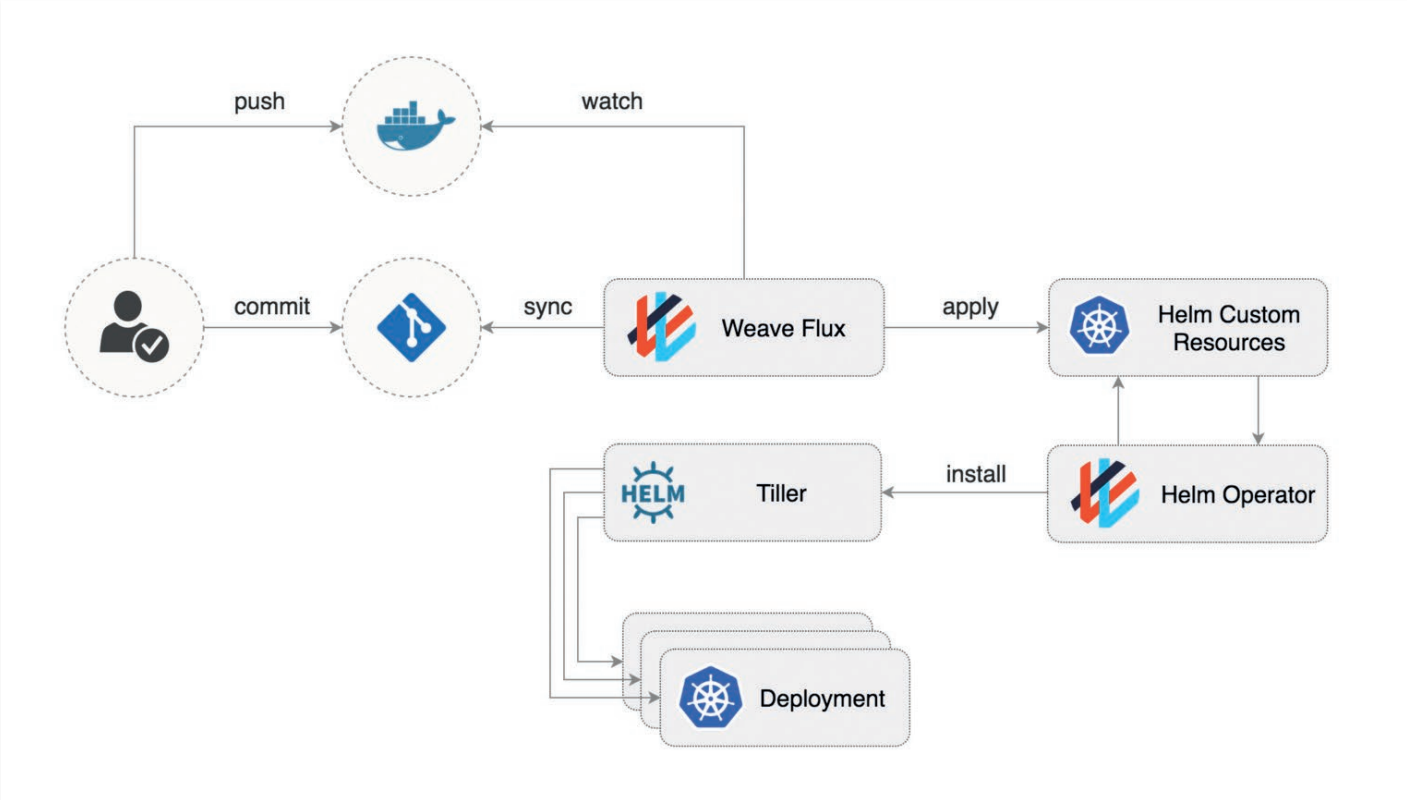

Abbildung 1: Flux mit Helm-Operator (© Weaveworks [2])

```
apiVersion: flux.weave.works/v1beta1
kind: HelmRelease
metadata:
  name: redis
  annotations:
    flux.weave.works/automated: "false"
spec:
  releaseName: redis
  chart:
    repository: https://kubernetes-charts.storage.googleapis.com/
    name: redis-ha
    version: 3.6.1
  values:
    exporter:
      enabled: true
    redis:
      masterGroupName: cacheMaster
```

Listing 2: Beispiel eines statischen HelmRelease

```
apiVersion: flux.weave.works/v1beta1
kind: HelmRelease
metadata:
  name: demo-service
  annotations:
    flux.weave.works/tag.chart-image: semver:~1.2
    flux.weave.works/automated: "true"
spec:
  values:
    image:
      repository: eu.gcr.io/project/demo-service
      tag: 1.2.0
    configuration:
      config_map_key_1: "config_map_value_1"
```

Listing 3: Automatisches Upgrade anhand semantischer Versionierung

Kubernetes arbeitet deklarativ

Kubernetes an sich ist ein großartiges System, jedoch ist es nicht einfach, den Zustand und die Version aller Ressourcen im Blick zu behalten. Verschiedene Stages mit unterschiedlichen Versionen und Konfigurationen der Fachanwendungen erschweren den Überblick weiter. Helm ist bereits eine große Hilfe, indem alle Ressourcen einer Anwendung zu einem sogenannten Chart geschnürt und gemeinsam verwaltet werden. Die Manifeste der Kubernetes-Ressourcen sind statisch. Helm fügt dank seiner Templating-Engine die Möglichkeit dynamischer Konfiguration hinzu.

Eine große Anzahl vom Helm-Projekt gepflegter Charts befinden sich im dortigen Git Repository und sofern eigene Charts existieren, werden diese vermutlich bereits ebenfalls in Git verwaltet. Wäre es nicht traumhaft, wenn das in Git Beschriebene auch dem tatsächlichen Zustand des Systems entspräche und Änderungen am System in Git ersichtlich würden? Das ist GitOps!

Flux: Der Delivery-Operator

Die Firma Weaveworks pflegt den Open Source Kubernetes Controller „Flux", der als Continuous Delivery Operator dient. Flux synchronisiert ein Git Repository mit dem eigenen Kubernetes Cluster. Ein Zugriff auf den Cluster von außen ist somit nicht nötig, was dessen Absicherung erleichtert. Die Installation von Flux ist dank des bereitgestellten Helm Chart denkbar einfach *(siehe Listing 1)*. Für eine Installation via Terraform eignet sich das Terraform HelmRelease gleichermaßen.

Flux erzeugt beim ersten Start ein Schlüsselpaar, mithilfe dessen man ihm den Zugriff auf das Git Repository gewähren kann. Den dafür nötigen öffentlichen Schlüssel erhält man entweder aus den Logs oder nach Installation des zugehörigen Kommandozeilen-Tools per „fluxctl identity". Durch regelmäßiges Polling (der Standard sind fünf Minuten) überwacht Flux das konfigurierte Repository und wendet alle gefundenen Kubernetes-Manifeste im Cluster an. Neben den zur Fachanwendung gehörenden können dies auch weitere allgemeine Ressourcen-Manifeste wie Istio Konfiguration oder Grafana Dashboards sein. Zur einfachen Nachvollziehbarkeit markiert Flux den zuletzt angewandten Commit mit einem Tag. Sollte der tatsächliche Stand in Kubernetes zum Beispiel durch manuelle Eingriffe abweichen, würde Flux diesen bei der nächsten Synchronisation wieder überschreiben.

Integration mit Helm

Richtig mächtig wird das Konstrukt, wenn man Helm hinzunimmt *(siehe Abbildung 1)*. Flux bietet einen optionalen Helm Operator an (bereitgestellt und konfiguriert durch die `helmOperator.*`-Values bei der Installation), der Hand in Hand mit dem Flux Operator zusammenarbeitet. Die durch Flux installierte Custom Resource Definition (CRD) „HelmRelease" ermöglicht die deklarative Installation eines Helm Chart. Referenzierte Charts können im GitOps Repository oder jedem anderen Git oder Helm Repository liegen. Ein HelmRelease wird, wie die anderen Kubernetes-Manifeste des GitOps Repository, durch Flux angelegt. Der Helm Operator installiert dann das Helm Chart. Values, die ein Chart konfigurieren, sind Teil des HelmRelease, womit die komplette Konfiguration der Anwendung an einer Stelle in Git gebündelt *(siehe Listing 2)* und nicht auf mehrere Tools verteilt ist.

Deployment-Automatisierung

Gehen wir auf eines der nützlichsten Features von Flux ein, das automatische Update von Helm Charts. Interessant sind dabei die Annotationen aus *Listing 3*:

- flux.weave.works/tag.chart-image: semver:~1.2
- flux.weave.works/automated: "true"

Erstere beschreibt ein Pattern für Image Tags, in unserem Fall möchten wir alle 1.2.x-Releases gemäß semantischer Versionierung verwenden. Die zweite Annotation weist Flux an, das Deploy-

```
bb2c260 (origin/master) Auto-release eu.gcr.io/project/demo-service:1.2.1
diff --git a/demo-service.yaml b/demo-service.yaml
index 8f32a81..be28faf 100644
--- a/demo-service.yaml
+++ b/demo-service.yaml
@@ -16,7 +16,7 @@ spec:
   values:
     image:
       repository: eu.gcr.io/project/demo-service
-      tag: 1.2.0
+      tag: 1.2.1
     container:
       replicaCount: 1
```

Listing 4: Von Flux erzeugter Git-Commit für das Update von 1.2.0 auf 1.2.1

ment zu automatisieren und neue, das heißt passende Versionen selbstständig auszuspielen. Alternativ zur semantischen Versionierung versteht Flux auch Glob-Muster *[3]* und reguläre Ausdrücke.

Diverse Varianten für die Angabe von Docker Image und Tag, die in vielen Helm Charts genutzt werden, erkennt Flux per Konvention (siehe *[4]*) und überwacht dann ebenfalls die Docker Registry. Wird dort ein neues Image mit entsprechendem Tag gefunden, erstellt Flux einen Commit, der das Image Tag im Manifest des HelmRelease ändert *(siehe Listing 4)*, und aktualisiert das HelmRelease im Kubernetes Cluster. Der Helm Operator führt daraufhin ein Upgrade des Helm Chart durch: Continuous Delivery.

Ein beispielhaftes Helm Chart für eine Fachanwendung samt zugehöriger Konfiguration könnte nun wie im *Listing 5* aussehen und vollständig durch das HelmRelease konfiguriert werden. Die ConfigMap sowie das SealedSecret (siehe weiter unten) werden komplett aus den Values des Chart *(siehe Listing 6)* gefüllt und sind im Manifest des Flux HelmRelease hinterlegt. Änderungen an der Konfiguration werden per Pull Request im Git Repository durchgeführt und unterliegen damit den gleichen Review-Prinzipien wie der Quellcode.

Geheimnisse in Git

Unsere gesamte Laufzeitumgebung ist nun in Git beschrieben. Etwas fehlt jedoch noch für ein rundes Konzept. Einige Konfigurationselemente wie Zugangsdaten und Passwörter können nicht im Klartext veröffentlicht werden. Hier eilt ein weiterer Kubernetes Controller zu Hilfe: SealedSecrets *[5]* *(siehe Abbildung 2)*.

Das SealedSecret ist ebenfalls eine CRD und enthält die Daten eines normalen Kubernetes Secret, jedoch asynchron verschlüsselt. Dazu erzeugt der SealedSecrets Controller ebenfalls ein Schlüsselpaar, dessen öffentlicher Teil Secrets mit dem eigenen CLI „kubeseal" verschlüsselt *(siehe Listing 7)*. Der private Teil verbleibt im Cluster und ist auch nur dort bekannt. Dadurch kann das SealedSecret bedenkenlos eingecheckt werden, die Entschlüsselung in ein normales Secret ist nur dem Controller im Kubernetes-Cluster möglich. Der private Teil des Schlüssels ist so das einzige außerhalb von Git zu hütende Geheimnis.

Deployment-Update bei Änderung der Konfiguration

Ein umfassendes Helm Chart wie im obigen Beispiel mit Anwendungskonfiguration in einer ConfigMap, bietet ein Stolperfalle: Helm wendet bei einem Upgrade nur die geänderten Ressourcen an, also

```
demo-service
+-- Chart.yaml
+-- templates
|   +-- ConfigMap.yaml
|   +-- Deployment.yaml
|   +-- HorizontalPodAutoscaler.yaml
|   +-- PodDisruptionBudget.yaml
|   +-- SealedSecret.yaml
|   +-- Service.yaml
+-- values.yaml
```

Listing 5: Struktur eines Helm-Charts für eine Fachanwendung

```
apiVersion: v1
kind: ConfigMap
metadata:
    name: "{{ .Chart.Name }}-env"
    namespace: {{ .Values.namespace }}
data:
    {{- toYaml .Values.configuration | nindent 4 }}
```

Listing 6: Kubernetes ConfigMap-Template

```
kubectl create secret generic demo-secret \
  --from-literal=password=root -o yaml \
  --dry-run > secret.yaml
kubeseal < secret.yaml > sealed-secret.yaml
```

Listing 7: Erzeugung eines neuen SealedSecret

bei Konfigurationsänderungen auch nur die ConfigMap. Um der Anwendung neue Konfigurationen mitzuteilen, ist im einfachsten Fall ein Neustart der Pods notwendig. Hier hilft als kleiner Trick *[6]*, die Checksumme der Konfigurationselemente in das Deployment zu schreiben, um diesen Neustart zu erwirken *(siehe Listing 8)*.

Deployment-Monitoring

Ein kleiner Wermutstropfen ist augenblicklich noch die Überwachung der GitOps Tools selbst. Flux und SealedSecrets bringen CLIs mit, die Einblicke in Zustand und Fehler erlauben. Darauf ein Monitoring aufzubauen, dürfte sich jedoch schwierig gestalten. Eine Möglichkeit ist es, die Log-Ausgaben der Controller auszuwerten, im Falle der Google-Cloud zum Beispiel mittels Stackdriver. Eine homogene Lösung lässt sich mithilfe eines weiteren Tools aus dem Hause Weaveworks realisieren: Kubediff *[7]* gleicht die tatsächliche Konfiguration in Kubernetes mit dem gewünschten Zustand im GitOps Repository ab und bietet darüber hinaus einen Prometheus-Export zur Integration in bestehendes Monitoring.

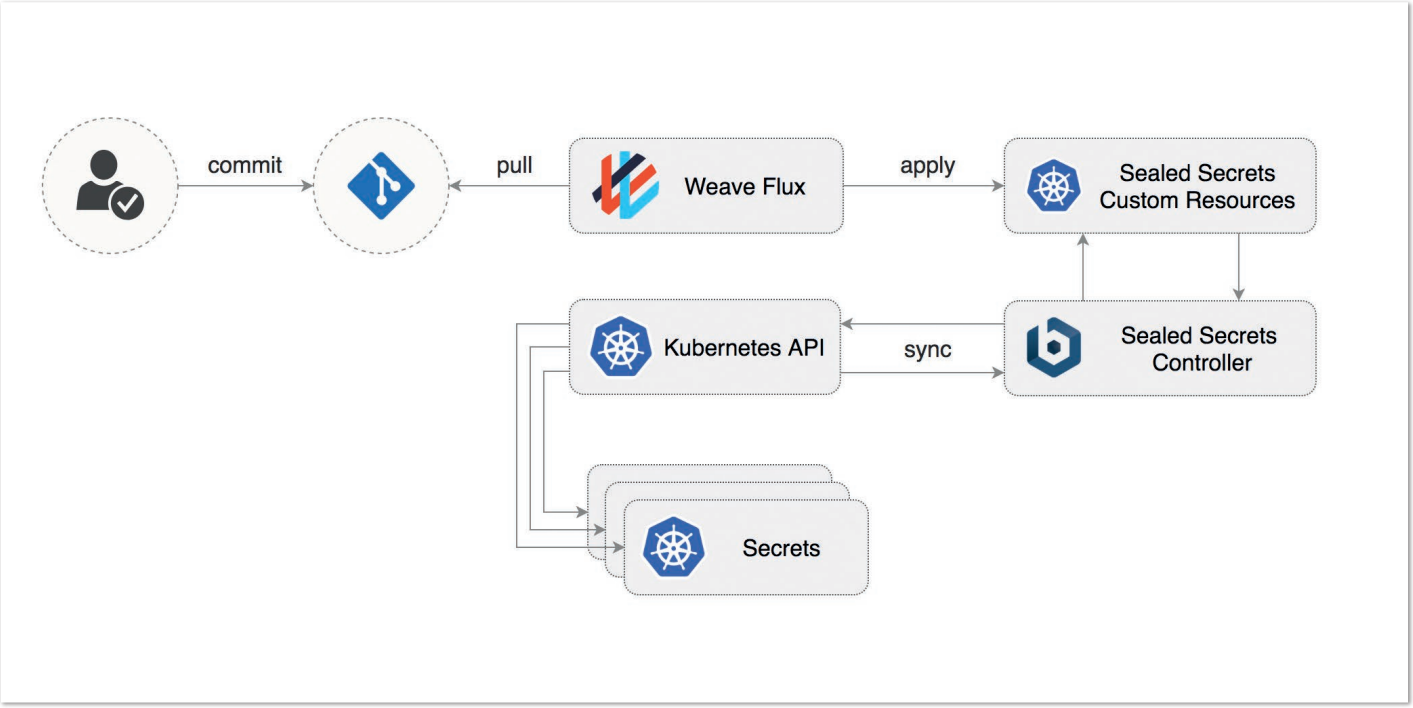

Abbildung 2: Integration mit SealedSecrets (© Weaveworks [2])

```
apiVersion: apps/v1
kind: Deployment
spec:
  template:
    metadata:
      annotations:
        checksum/config: \
{{ include (print $.Template.BasePath "/ConfigMap.yaml") . | sha256sum }}
        checksum/secret: \
{{ include (print $.Template.BasePath "/SealedSecret.yaml") . | sha256sum }}
```

Listing 8: Deployment-Template mit Checksummen für Konfigurationselemente

Fazit

Durch GitOps lässt sich mit nur wenig Konfiguration eine flexible und zuverlässige Continuous-Delivery-Lösung gestalten. Dank der Feedback-Schleife des Delivery Operators wird Git zur alleinigen Quelle der Wahrheit und zum zentralen Element der Entwicklungs- und Betriebsprozesse. Etablierte Review- und Freigabemechanismen können gleichermaßen angewandt werden, wodurch auch Konfigurationsänderungen nachvollziehbar dokumentiert sind.

Quellen

[1] Alexis Richardson (2017): GitOps - Operations by Pull Request. *https://www.weave.works/blog/gitops-operations-by-pull-request*

[2] Weaveworks: Managing Helm releases the GitOps way. *https://www.weave.works/blog/managing-helm-releases-the-gitops-way*

[3] Wikipedia: glob. *https://en.wikipedia.org/wiki/Glob_(programming)*

[4] Flux Reference: Upgrading images in a HelmRelease using Flux. *https://docs.fluxcd.io/en/latest/references/helm-operator-integration.html#upgrading-images-in-a-helmrelease-using-flux*

[5] SealedSecrets. *https://github.com/bitnami-labs/sealed-secrets*

[6] Chart Development Tips and Tricks. *https://github.com/helm/helm/blob/master/docs/charts_tips_and_tricks.md*

[7] Weaveworks: Kubediff. *https://github.com/weaveworks/kubediff*

Bernd Stübinger, Florian Heubeck
Java User Group Ingolstadt e.V.
info@jug-in.bayern

Bernd und Florian sind Mitgründer und Organisatoren der JUG Ingolstadt. Für ihren Arbeitgeber MediaMarktSaturn erstellen sie ganzheitliche Softwarelösungen und geben ihre Erfahrungen mit neuen Tools und Konzepten gern weiter.