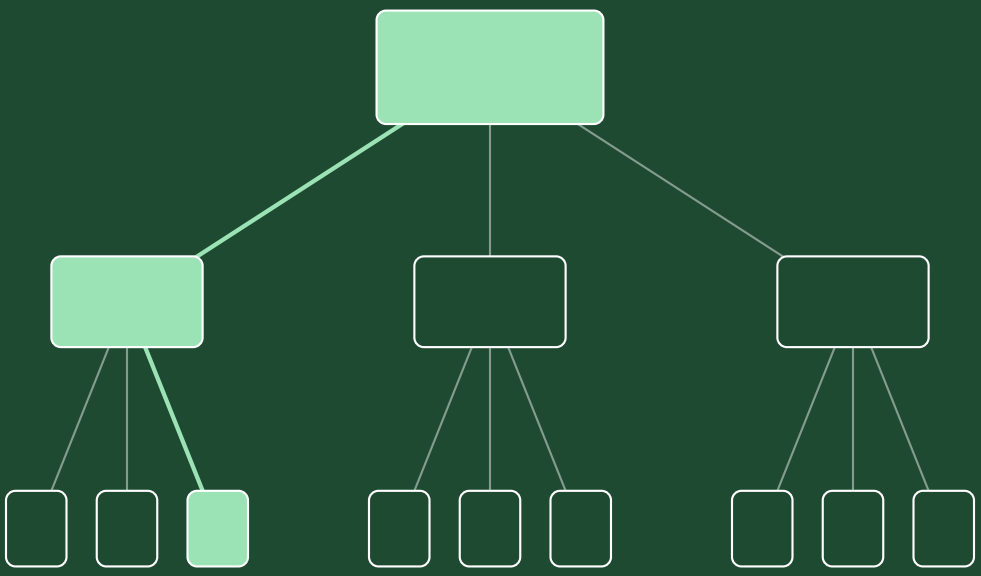
nyrvex.com

Helm Charts

Hands-On

Kubernetes-Anwendungen paketieren,
ausliefern und betreiben



Mit wenig Text fit fürs Interview

Helm Charts

Hands-On

Kubernetes-Anwendungen paketieren, ausliefern und betreiben

Mit wenig Text fit fürs Interview

Thomas Zachmann

Helm Charts Hands-On. Kubernetes-Anwendungen paketieren, ausliefern und betreiben.
Stand: November 2026. © 2026 Thomas Zachmann. Lizenz: CC BY-ND 4.0.

Geschrieben für Helm 4 (Stand der Befehle: Helm 4.3). Die Befehle, Charts und Beispielausgaben in diesem Buch wurden nicht ausgeführt. Ausgaben sind beispielhaft, und Details können sich je nach Version unterscheiden. Probiere alles zuerst in einem Testcluster aus, nie direkt in Produktion. Domains wie `example.com` sind Platzhalter.

Schriften: Charter, Fira Sans, JetBrains Mono.

Inhalt

So liest du dieses Buch 6

Der Start **9**
1 Was ist Helm? 10
2 Chart, Release, Repository 12
3 Das Labor 14
4 Der erste Release 16

Charts bauen **19**
5 Der Chart-Aufbau 20
6 Chart.yaml 22
7 values.yaml und Vorrang 24
8 Die eingebauten Objekte 27

Templates denken **29**
9 Template-Grundlagen 30
10 Whitespace 32
11 if, with, range 34
12 Variablen und Scope 36
13 define, include, _helpers.tpl 38
14 Sprig: die wichtigsten Funktionen 40
15 tpl, required, lookup 42
16 Dateien einlesen 44

Fortgeschrittene Charts **47**
17 values.schema.json 48
18 Dependencies und Subcharts 50
19 Library Charts 53
20 Hooks 55
21 Chart-Tests 57
22 CRDs 59

Betrieb **61**
23 install, upgrade, rollback 62
24 Wo Helm seine Releases speichert 65
25 3-Way-Merge und Server-Side Apply 67
26 Was Helm 4 ändert 70
27 OCI-Registries 72
28 Signieren und Provenance 74
29 Secrets 76

Qualität und Auslieferung **79**
30 Lint und Rendern 80
31 helm-unittest 82
32 CI-Pipeline 84
33 GitOps: Argo CD und Flux 86
34 Helm oder Kustomize? 89

Troubleshooting **91**
35 Die Methode 92
36 pending-upgrade 94
37 Immutable Fields 97
38 Ownership und Konflikte 99

Das Interview **103**

39 Die Fragen 104
40 Aufgaben 108
41 Spickzettel 111
Stichwörter 114

So liest du dieses Buch

Du hast bald ein Interview und musst Helm können. Nicht irgendwann, sondern bald. Dieses Buch ist genau dafür gebaut: wenig Text, viele kleine Beispiele, und zu jedem Thema die Frage, die dir wirklich gestellt wird.

Die Regeln

- **Tippe alles selbst.** Nichts kopieren. Deine Finger lernen mit.
- **Jeder Befehl einzeln.** Du tippst, du schaust, was passiert.
- **Rendern ist kostenlos.** `helm template` braucht keinen Cluster. Damit siehst du jedes Template-Beispiel sofort.
- **Lies das Ergebnis.** Am Ende zählt nur das YAML, das Kubernetes bekommt.

> **MERKE**
>
> Die Befehle, Charts und Ausgaben in diesem Buch wurden nicht ausgeführt. Die Ausgaben sind gekürzte Beispiele. Deine Texte und Zahlen werden abweichen.

Die Kästen

> **DAS BILD IM KOPF**
>
> *Ein Vergleich aus dem Alltag. Wenn du den behältst, kannst du das Thema herleiten.*

```
helm template web ./web
```

Eselsbrücke: Ein Befehl zum Tippen, mit Merkhilfe darunter.

> **STOLPERFALLE**
>
> Der Fehler, den fast alle machen. Und den Interviewer gern abfragen.

IM INTERVIEW

Eine typische Frage?

Eine Antwort in zwei, drei Sätzen. So kurz darf sie sein.

Dein Lernplan

Du hast …	Dann lies …
1 Abend	Kapitel 1, 2, 5, 7, 11, 13, 23, 25, dann Kapitel 39 (die Fragen) und 41 (Spickzettel).
2 Tage	Tag 1: Teil I bis III. Tag 2: Kapitel 18, 20, 22 bis 26, 36 bis 38, dann Kapitel 39.
1 Woche	Tag 1: Teil I und II. Dann ein Teil pro Tag. Tag 6: die Aufgaben aus Kapitel 40. Tag 7: Fragen wiederholen.

TEIL I

Der Start

Was Helm ist, wie es denkt und wie du deinen ersten Release installierst.

1

Was ist Helm?

Der Paketmanager für Kubernetes.

Eine Anwendung in Kubernetes besteht aus vielen YAML-Dateien: Deployment, Service, Ingress, ConfigMap und mehr. Helm packt sie zu einem Paket zusammen. Seine Kernidee passt in eine Zeile:

Manifeste = Funktion(Templates, Werte)

Du schreibst die Templates einmal. Für jede Umgebung änderst du nur die Werte.

Das Problem

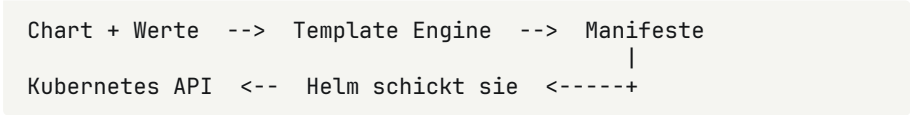
Ohne Helm hast du drei Kopien deiner Manifeste für Dev, Staging und Prod. Sie unterscheiden sich in fünf Zeilen, und mit der Zeit in viel mehr. Außerdem weiß Kubernetes nicht, welche Objekte zusammengehören oder was gestern installiert war.

Helm löst beides:

- **Paketieren:** Templates und Standardwerte werden zu einem Chart.
- **Konfigurieren:** Pro Umgebung kommen eigene Werte hinzu.
- **Lebenszyklus:** Installieren, aktualisieren, zurückrollen und entfernen als eine Einheit.

> **DAS BILD IM KOPF**
>
> *Helm ist wie `apt` für Kubernetes. Ein Chart ist das Paket, der Cluster ist der Rechner, und `helm upgrade` ist das Update. Dazu kommt ein Serienbrief: Die Vorlage bleibt gleich, nur die Empfänger-Daten wechseln.*

So arbeitet Helm



```
                                                  |
Secret im Cluster  <--  Helm merkt sich den Release
```

Helm läuft als Programm auf deinem Rechner oder im CI. Es gibt keinen Server im Cluster. Helm spricht direkt mit der Kubernetes-API und nutzt dein `kubeconfig` samt deinen Rechten.

> **MERKE**
>
> Helm 2 hatte einen Server im Cluster, Tiller. Er war ein Sicherheitsproblem und ist seit Helm 3 weg. Das wird im Interview gern gefragt.

Die Versionen

Version	Status im November 2026
Helm 4	Aktuell, seit November 2025. Darauf baut dieses Buch auf.
Helm 3	Nur noch Sicherheitsupdates, bis 10. Februar 2027.
Helm 2	Längst eingestellt.

IM INTERVIEW

Was ist Helm, und welches Problem löst es?

Der Paketmanager für Kubernetes. Er macht aus Manifesten wiederverwendbare Templates, trennt Konfiguration in Werte und verwaltet eine Anwendung als Einheit, mit Historie und Rollback.

IM INTERVIEW

Warum wurde Tiller entfernt?

Tiller lief mit weiten Rechten im Cluster und umging damit die Rechte der einzelnen Nutzer. Seit Helm 3 arbeitet der Client direkt mit der API und den RBAC-Rechten des Nutzers.

DAS NIMMST DU MIT

- Manifeste = Funktion(Templates, Werte).
- Helm läuft clientseitig, ohne Server im Cluster.
- Helm 4 ist aktuell.

2

Chart, Release, Repository

Drei Wörter, und du verstehst jedes Helm-Gespräch.

Begriff	Bedeutung	Vergleich
Chart	Das Paket: Templates, Standardwerte, Metadaten.	Rezept
Release	Eine installierte Instanz eines Charts im Cluster.	Gekochtes Gericht
Repository	Ein Ort, an dem Charts liegen.	Kochbuch-Regal
Revision	Ein Versionsstand eines Releases.	Nummer im Logbuch

Ein Chart kann mehrfach installiert werden. Jede Installation ist ein eigener Release mit eigenem Namen. So laufen `shop-a` und `shop-b` aus demselben Chart.

Jedes `upgrade` und jedes `rollback` erzeugt eine neue Revision. Ein Rollback geht also nicht „zurück“, sondern schreibt den alten Stand als neue Nummer ins Logbuch.

```
Revision 1  install    v1.0
Revision 2  upgrade    v1.1
Revision 3  rollback   v1.0 (Inhalt von Revision 1)
```

Zwei Versionen in Chart.yaml

- `version` ist die Version des **Charts**.
- `appVersion` ist die Version der **Anwendung**, die darin steckt.

Beide sind unabhängig. Änderst du nur ein Template, steigt `version`, während `appVersion` gleich bleibt.

Woher Charts kommen

- **OCI-Registry:** wie Container-Images. Der heutige Standard, siehe Kapitel 27.
- **HTTP-Repository:** ein Webserver mit einer `index.yaml`.
- **Lokal:** ein Ordner oder eine `.tgz`-Datei.

- **Artifact Hub:** eine Suchmaschine für öffentliche Charts.

IM INTERVIEW

Was ist der Unterschied zwischen Chart und Release?

Der Chart ist das Paket, der Release ist seine laufende Instanz im Cluster. Ein Chart kann als viele Releases installiert sein, auch im selben Namespace, solange die Namen sich unterscheiden.

IM INTERVIEW

`version` oder `appVersion`?

`version` ist die SemVer-Version des Charts und bestimmt das Paket. `appVersion` beschreibt die enthaltene Anwendung und ist nur eine Information, die Templates oft als Standard-Image-Tag nutzen.

DAS NIMMST DU MIT

- Chart ist das Rezept, Release das Gericht.
- Jede Änderung ergibt eine neue Revision.
- `version` gehört zum Chart, `appVersion` zur Anwendung.

3

Das Labor

Ein lokaler Cluster und das Helm-Programm. Mehr brauchst du nicht.

Für dieses Buch reicht ein Laptop. Du brauchst Docker, `kubectl`, `kind` und `helm`.

```
kind create cluster --name helm-labor
```

Eselsbrücke: **kind** heißt **K**ubernetes **in** **D**ocker: ein Cluster in Containern.

```
kubectl get nodes
```

Eselsbrücke: Antwortet dein Cluster?

```
helm version
```

Eselsbrücke: Du fragst Helm nach seinem Ausweis. Hier sollte eine 4 vorne stehen.

Wen spricht Helm an?

Helm nutzt denselben Kontext wie `kubectl`.

```
kubectl config current-context
```

Eselsbrücke: Bin ich im richtigen Cluster? Das fragst du dich vor jedem `upgrade`.

```
helm env
```

Eselsbrücke: Zeigt Pfade für Cache, Konfiguration und Daten.

> **STOLPERFALLE**
>
> Helm nimmt ohne Warnung den aktuellen Kontext. Wer gerade noch in Produktion war, installiert dort. Setze in Skripten immer `--kube-context` und `--namespace` ausdrücklich.

Ein Chart zum Spielen

```
helm create web
```

Eselsbrücke: **create** legt einen fertigen Beispiel-Chart im Ordner `web` an.

```
helm template web ./web
```

Eselsbrücke: Rendere nur, schicke nichts ins Cluster. Dein wichtigster Befehl beim Lernen.

DAS NIMMST DU MIT

- `kind`, `kubectl`, `helm`.
- Prüfe den Kontext vor jedem Eingriff.
- `helm template` zeigt das Ergebnis ohne Cluster.

4

Der erste Release

Installieren, ansehen, ändern, zurückrollen, entfernen.

Installieren

```
helm install web ./web --namespace demo --create-namespace
```

Eselsbrücke: Lies es als Satz: Installiere als Release `web` den Chart aus `./web` in den Namespace `demo`.

```
NAME: web
NAMESPACE: demo
STATUS: deployed
REVISION: 1
```

Ansehen

```
helm list -n demo
```

Eselsbrücke: Welche Releases gibt es hier? `-A` zeigt alle Namespaces.

```
helm status web -n demo
```

Eselsbrücke: Wie steht der Release, und was sagen die Notes?

```
helm get values web -n demo
```

Eselsbrücke: Welche Werte habe **ich** gesetzt?

```
helm get manifest web -n demo
```

Eselsbrücke: Was hat Helm wirklich an den Cluster geschickt?

Ändern

```
helm upgrade web ./web -n demo --set replicaCount=3
```

Eselsbrücke: Neue Werte, neue Revision. Helm ändert nur, was sich unterscheidet.

```
helm history web -n demo
```

Eselsbrücke: Das Logbuch aller Revisionen.

Zurückrollen

```
helm rollback web 1 -n demo
```

Eselsbrücke: Zurück auf den Inhalt von Revision 1. Ohne Nummer geht es zur vorherigen.

Entfernen

```
helm uninstall web -n demo
```

Eselsbrücke: Entfernt die Objekte des Releases und seine Historie.

> **STOLPERFALLE**
>
> `helm uninstall` löscht auch die Historie. Danach gibt es kein Rollback mehr. Mit `--keep-history` bleibt das Logbuch erhalten. Außerdem bleiben PVCs aus `volumeClaimTemplates` eines StatefulSets liegen, weil sie nicht zum Release gehören.

Fremde Charts nutzen

```
helm show values oci://registry.example.com/charts/web
```

Eselsbrücke: Lies zuerst, was sich einstellen lässt. Mit `helm show all` kommt alles.

```
helm pull oci://registry.example.com/charts/web --version
1.2.0 --untar
```

Eselsbrücke: Hol den Chart auf die Platte und schau hinein, bevor du ihm vertraust.

IM INTERVIEW

Was passiert bei `helm install`?

Helm lädt den Chart, führt Werte zusammen, rendert die Templates und prüft sie. Dann schickt es die Manifeste an die API und speichert den Release als Secret im Namespace.

DAS NIMMST DU MIT

- `install`, `list`, `status`, `upgrade`, `history`, `rollback`, `uninstall`.
- `helm get` zeigt, was im Release gespeichert ist.
- Lies fremde Charts, bevor du sie installierst.

TEIL II

Charts bauen

Ordner, Metadaten, Werte und die eingebauten Objekte.

5

Der Chart-Aufbau

Ein Chart ist nur ein Ordner mit festen Namen.

```
web/
├── Chart.yaml          Name, Version, Abhängigkeiten
├── Chart.lock          Fest eingefrorene Abhängigkeiten
├── values.yaml         Standardwerte
├── values.schema.json  Prüfregeln für Werte (optional)
├── charts/             Abhängigkeiten
├── crds/               CRDs (optional)
├── templates/
│   ├── NOTES.txt       Text nach der Installation
│   ├── _helpers.tpl    Eigene Bausteine
│   ├── deployment.yaml
│   ├── service.yaml
│   └── tests/          Chart-Tests
├── .helmignore         Was nicht ins Paket soll
└── README.md
```

Regel	Bedeutung
Alles in `templates/`	wird gerendert und installiert.
Dateien mit `_` vorne	erzeugen kein Manifest. Sie enthalten nur Bausteine.
`NOTES.txt`	wird gerendert und nach der Installation angezeigt.
`templates/tests/`	enthält Pods, die `helm test` startet.

Ein Template sollte genau ein Kubernetes-Objekt enthalten. Der Dateiname nennt die Art, zum Beispiel `deployment.yaml`.

> **DAS BILD IM KOPF**
>
> *Ein Chart ist wie ein Koffer mit festen Fächern: Metadaten ins Außenfach, Vorlagen ins Hauptfach, Kleinteile in die Tasche mit dem Unterstrich.*

Paketieren

```
helm package ./web
```

Eselsbrücke: Packt den Ordner in `web-1.0.0.tgz`. Der Name stammt aus `Chart.yaml`.

```
helm show chart web-1.0.0.tgz
```

Eselsbrücke: Lies die Metadaten aus dem Paket.

IM INTERVIEW

Wozu dient `_helpers.tpl`?

Dort stehen benannte Templates, die der Chart wiederverwendet, etwa für Namen und Labels. Der Unterstrich sagt Helm: Diese Datei erzeugt kein Manifest.

DAS NIMMST DU MIT

- `Chart.yaml`, `values.yaml`, `templates/`.
- Der Unterstrich bedeutet: nur Bausteine.
- Ein Objekt pro Datei.

6

Chart.yaml

Der Ausweis des Pakets.

Chart.yaml

```yaml
apiVersion: v2
name: web
description: Ein kleiner Webdienst
type: application
version: 1.4.2
appVersion: "2.7.0"
kubeVersion: ">=1.30.0-0"
maintainers:
  - name: Team Plattform
    email: plattform@example.com
```

Feld	Bedeutung
`apiVersion`	Das Chart-Format. `v2` ist der Standard.
`name`	Der Name des Charts.
`type`	`application` oder `library`, siehe Kapitel 19.
`version`	Chart-Version, zwingend SemVer.
`appVersion`	Version der Anwendung. Am besten in Anführungszeichen.
`kubeVersion`	Welche Kubernetes-Versionen der Chart verträgt.
`dependencies`	Andere Charts, siehe Kapitel 18.

SemVer für Charts

- **Major:** Bruch. Nutzer müssen etwas tun, zum Beispiel Werte umbenennen.
- **Minor:** neues Feature, abwärtskompatibel.
- **Patch:** Fehlerbehebung.

> **STOLPERFALLE**
>
> `appVersion: 2.10` ohne Anführungszeichen liest YAML als Zahl, also `2.1`. Schreibe Versionen immer als String.

MERKE

Es gibt zusätzlich experimentelle Charts mit `apiVersion: v3`. Sie sind hinter `HELM_EXPERIMENTAL_CHART_V3=1` versteckt und können sich noch ändern. Für die Praxis bleibt `v2` richtig, und bestehende Charts laufen unter Helm 4 weiter.

IM INTERVIEW

Wann erhöhst du welche Stelle der Chart-Version?

Bei jeder Änderung am Chart mindestens Patch. Minor bei neuen Optionen, Major, wenn Nutzer etwas anpassen müssen, etwa bei umbenannten Werten oder entfernten Optionen.

DAS NIMMST DU MIT

- `apiVersion: v2` ist der Standard.
- `version` ist SemVer.
- Versionen in Anführungszeichen.

7

values.yaml und Vorrang

Wer gewinnt, wenn mehrere Quellen denselben Wert setzen?

`values.yaml` ist die Standardkonfiguration und die wichtigste Dokumentation deines Charts.

values.yaml

```yaml
# Anzahl der Pods
replicaCount: 2

image:
  repository: registry.example.com/web
  # Standard ist appVersion aus Chart.yaml
  tag: ""

service:
  type: ClusterIP
  port: 80
```

Die Reihenfolge

Von schwach nach stark, also das Untere gewinnt:

1. `values.yaml` des Charts.
2. Werte des übergeordneten Charts, wenn es ein Subchart ist.
3. Dateien mit `-f`, die rechte schlägt die linke.
4. `--set` und seine Varianten, das rechte schlägt das linke.

```
helm upgrade web ./web -f prod.yaml -f geheim.yaml --set replicaCount=5
```

Eselsbrücke: Lies von links nach rechts: Wer zuletzt spricht, gewinnt.

> **DAS BILD IM KOPF**
>
> *Klebezettel auf einem Plan: Jede neue Schicht klebt über die alte. Am Ende sieht man nur den obersten Zettel.*

Die Geschwister von `--set`

Flag	Wirkung
`--set a=1`	Typ wird erraten, hier eine Zahl.
`--set-string a=1`	Immer ein String.
`--set-literal a=1,2`	Der ganze Text ist der Wert, ohne Sonderzeichen-Logik.
`--set-json 'a={"b":1}'`	Wert als JSON.
`--set-file a=datei.txt`	Wert aus dem Inhalt einer Datei.

Pfade für verschachtelte Werte: `--set image.tag=2.7.0`. Listen-Elemente: `--set hosts[0]=a.example.com`.

Wie Werte zusammenlaufen

- **Maps** werden zusammengeführt.
- **Listen** werden komplett ersetzt, nie ergänzt.
- `null` löscht einen Standardwert.

> **STOLPERFALLE**
>
> Du willst eine Umgebungsvariable ergänzen und schreibst die Liste `env` in `prod.yaml`. Alle Einträge aus `values.yaml` sind danach weg. Wer ergänzbare Konfiguration braucht, nimmt Maps statt Listen.

Beim Upgrade

Flag	Wirkung
Ohne Flag	Chart-Standards plus das, was du jetzt angibst.
`--reuse-values`	Alte Werte des Releases, dazu deine neuen. Neue Chart-Standards fehlen.
`--reset-values`	Nur die Standards des Charts.
`--reset-then-reuse-values`	Neue Standards, darüber die alten Werte, darüber deine neuen.

> **MERKE**
>
> Am sichersten ist, bei jedem Upgrade alle Werte aus Dateien anzugeben. Dann steht der Sollzustand im Git, nicht im Kopf. `--set` nimmst du nur für Dinge wie einen Image-Tag aus dem CI.

IM INTERVIEW

In welcher Reihenfolge greifen Values?

`values.yaml` des Charts ist am schwächsten. Darüber liegen Werte des Eltern-Charts, dann Dateien mit `-f` von links nach rechts, dann `--set` von links nach rechts.

IM INTERVIEW

Warum ist `--reuse-values` riskant?

Es nimmt die Werte des letzten Releases statt der neuen Chart-Standards. Bringt ein neuer Chart einen neuen Wert mit, fehlt er, und das Template scheitert oder rendert leer.

DAS NIMMST DU MIT

- `values.yaml`, dann `-f`, dann `--set`. Rechts gewinnt.
- Maps mergen, Listen ersetzen.
- Werte in Dateien im Git, nicht im Kopf.

8

Die eingebauten Objekte

Was jedes Template ohne dein Zutun kennt.

Objekt	Beispiele
`.Values`	Alle zusammengeführten Werte.
`.Release`	`.Name`, `.Namespace`, `.Revision`, `.IsInstall`, `.IsUpgrade`, `.Service`
`.Chart`	Alles aus `Chart.yaml`: `.Name`, `.Version`, `.AppVersion`
`.Capabilities`	`.KubeVersion`, `.APIVersions.Has "..."`
`.Template`	`.Name`, `.BasePath`
`.Files`	Zugriff auf weitere Dateien im Chart.

templates/configmap.yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-config
  namespace: {{ .Release.Namespace }}
data:
  chart: "{{ .Chart.Name }}-{{ .Chart.Version }}"
  revision: {{ .Release.Revision | quote }}
```

Nach Cluster-Fähigkeiten fragen

```
{{- if .Capabilities.APIVersions.Has
    "gateway.networking.k8s.io/v1" }}
# Eine HTTPRoute anlegen
{{- end }}
```

> **STOLPERFALLE**
>
> `.Capabilities` fragt bei `helm template` keinen Cluster. Es nutzt eingebaute Standardwerte. Mit `--api-versions` und `--kube-version` gibst du sie ausdrücklich vor. Sonst kann dein CI etwas anderes rendern als der Cluster.

IM INTERVIEW

Wie machst du einen Chart für mehrere Kubernetes-Versionen fit?

Mit `.Capabilities`: Das Template fragt, ob eine API vorhanden ist, und rendert passend. `kubeVersion` in `Chart.yaml` sperrt nicht unterstützte Versionen aus.

DAS NIMMST DU MIT

- `.Values`, `.Release`, `.Chart`, `.Capabilities`, `.Template`, `.Files`.
- `.Release.IsUpgrade` unterscheidet Install und Upgrade.
- `helm template` kennt deinen Cluster nicht.

TEIL III

Templates denken

Go-Templates, Whitespace, Scope, Bausteine. Das Fundament jedes Interviews.

9

Template-Grundlagen

Doppelte Klammern, Punkte und Pipes.

Helm nutzt die Template-Sprache von Go, erweitert um die Bibliothek **Sprig**. Alles zwischen doppelten geschweiften Klammern ist eine **Action**.

```
name: {{ .Release.Name }}
image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
```

Der Punkt

Der Punkt ist der aktuelle Kontext. Oben im Template ist das die Wurzel mit allen Objekten. `.Values.image.tag` heißt also: von hier aus zu `Values`, zu `image`, zu `tag`.

Pipes

Wie in der Shell fließt ein Wert von links nach rechts durch Funktionen:

```
env: {{ .Values.env | default "prod" | upper | quote }}
```

Das Gleiche ohne Pipe, von innen nach außen gelesen:

```
env: {{ quote (upper (default "prod" .Values.env)) }}
```

Der Wert aus der Pipe landet immer als **letztes** Argument. Darum steht bei `default` zuerst der Ersatzwert.

> **DAS BILD IM KOPF**
>
> *Ein Fließband: Der Wert kommt links herein, jede Station bearbeitet ihn, und rechts fällt das Ergebnis heraus.*

Warum `quote`?

YAML rät Typen. `true`, `1.20` und `on` sind ohne Anführungszeichen keine Strings.

STOLPERFALLE

`value: {{ .Values.debug }}` ergibt für eine Umgebungsvariable einen Fehler, weil Kubernetes dort einen String verlangt und YAML `true` als Boolean liest. Setze bei Strings immer `| quote`. Zahlen wie Ports lässt du dagegen ohne.

Rendern als Kontrolle

```
helm template web ./web --show-only templates/service.yaml
```

Eselsbrücke: Zeig nur diese eine Datei. `-s` ist die Kurzform.

```
helm template web ./web --set image.tag=2.7.0 --debug
```

Eselsbrücke: `--debug` zeigt mehr und gibt auch kaputtes YAML aus, damit du den Fehler siehst.

IM INTERVIEW

Was bedeutet der Punkt in einem Helm-Template?

Der aktuelle Kontext. Auf oberster Ebene ist es die Wurzel mit `.Values`, `.Release` und so weiter. `range` und `with` ändern ihn.

DAS NIMMST DU MIT

- Action, Punkt, Pipe.
- Der Pipe-Wert ist das letzte Argument.
- Strings immer mit `quote`.

10

Whitespace

YAML ist pingelig. Ein Minuszeichen rettet dich.

Eine Action lässt ihre Zeile stehen, auch wenn sie nichts ausgibt. Der Rest bleibt als Leerzeile.

Schreibweise	Wirkung
`{{- ...`	Entfernt Leerraum **links**, samt Zeilenumbruch davor.
`... -}}`	Entfernt Leerraum **rechts**, samt Zeilenumbruch danach.

```
metadata:
  labels:
    {{- if .Values.team }}
    team: {{ .Values.team }}
    {{- end }}
```

Das Minus frisst den Zeilenumbruch vor der Action. So bleiben keine Leerzeilen übrig.

> **DAS BILD IM KOPF**
>
> *Das Minus ist ein Staubsauger, der in eine Richtung zeigt. Er saugt alle Leerzeichen und Umbrüche auf dieser Seite weg, bis er auf Text trifft.*

> **STOLPERFALLE**
>
> `{{- .Values.name -}}` frisst auf beiden Seiten. Steht die nächste Zeile direkt dahinter, klebt sie an der vorherigen, und das YAML ist kaputt. Nimm rechts nur dann ein Minus, wenn du das willst.

Blöcke einrücken

Für ganze Strukturen nimmst du `toYaml` und `nindent`:

```
    spec:
      containers:
        - name: web
```

```
          resources:
            {{- toYaml .Values.resources | nindent 12 }}
```

`nindent 12` beginnt mit einem Zeilenumbruch und rückt jede Zeile um 12 Stellen ein. Die Zahl entspricht der Einrückung des Blocks in der Zieldatei.

Funktion	Wirkung
`indent 4`	Rückt jede Zeile ein.
`nindent 4`	Wie `indent`, aber mit Zeilenumbruch davor.

IM INTERVIEW

Warum nutzt man `nindent` statt `indent`?

Mit `nindent` kann die Action links stehen, und das `{{-` entfernt den Umbruch und die Einrückung davor. Das Ergebnis beginnt sauber in einer neuen Zeile, und das Template bleibt lesbar.

DAS NIMMST DU MIT

- `{{-` frisst links, `-}}` frisst rechts.
- `toYaml` plus `nindent` für Blöcke.
- Fehlersuche: `helm template --debug`.

11

if, with, range

Bedingungen, Kontextwechsel und Schleifen.

if

```
{{- if .Values.ingress.enabled }}
apiVersion: networking.k8s.io/v1
kind: Ingress
# ...
{{- end }}
```

Falsch sind: `false`, `0`, der leere String, `null` und leere Listen oder Maps. Alles andere ist wahr.

Vergleiche sind Funktionen und stehen vorne: `eq`, `ne`, `lt`, `gt`, `and`, `or`, `not`.

```
{{- if and .Values.ingress.enabled (eq .Values.env "prod") }}
```

> **STOLPERFALLE**
>
> Ein Wert von `0` zählt als falsch. Nutzt du `default 3 .Values.replicas`, wird auch eine bewusst gesetzte 0 zu 3. Das Gleiche gilt für `false`. Prüfe in solchen Fällen mit `hasKey` oder `kindIs`.

with

`with` setzt den Punkt um und überspringt den Block, wenn der Wert leer ist.

```
{{- with .Values.nodeSelector }}
nodeSelector:
  {{- toYaml . | nindent 2 }}
{{- end }}
```

Das ist die saubere Form für optionale Blöcke.

range

`range` läuft über Listen und Maps. Dabei wird der Punkt zum aktuellen Element.

values.yaml

```yaml
hosts:
  - a.example.com
  - b.example.com
env:
  LOG_LEVEL: info
  REGION: eu
```

```
rules:
{{- range .Values.hosts }}
  - host: {{ . | quote }}
{{- end }}
env:
{{- range $name, $wert := .Values.env }}
  - name: {{ $name }}
    value: {{ $wert | quote }}
{{- end }}
```

Maps laufen in alphabetischer Reihenfolge. Das macht die Ausgabe stabil, und ein Upgrade sieht keine falschen Änderungen.

IM INTERVIEW

Was ist der Unterschied zwischen `if` und `with`?

Beide prüfen auf leer. `with` setzt zusätzlich den Punkt auf den Wert. Das spart Schreibarbeit, aber innerhalb des Blocks kommst du an `.Release` nur noch über `$`.

DAS NIMMST DU MIT

- Leere Werte, 0 und `false` sind falsch.
- `with` für optionale Blöcke.
- `range` ändert den Punkt.

12

Variablen und Scope

Das Dollarzeichen bringt dich zurück nach Hause.

Innerhalb von `range` und `with` zeigt der Punkt woanders hin. Dieser Fehler kommt daher:

```
{{- range .Values.hosts }}
  host: {{ . }}.{{ .Release.Namespace }}  # Fehler
{{- end }}
```

Der Punkt ist hier ein String, und ein String hat kein `.Release`.

Die Lösung: `$`

`$` zeigt immer auf die Wurzel des Templates.

```
{{- range .Values.hosts }}
  host: {{ . }}.{{ $.Release.Namespace }}
{{- end }}
```

> **DAS BILD IM KOPF**
>
> *Der Punkt ist dein aktueller Standort. Das Dollarzeichen ist die Zuhause-Taste im Navi: Egal wo du bist, sie bringt dich zum Anfang.*

Eigene Variablen

```
{{- $name := include "web.fullname" . }}
{{- range .Values.ports }}
  - name: {{ $name }}-{{ .name }}
    port: {{ .port }}
{{- end }}
```

`:=` legt eine Variable an, `=` weist neu zu.

IM INTERVIEW

Was tust du, wenn du in `range` auf `.Release` zugreifen musst?

Ich nehme `$.Release`, denn `$` zeigt auf die Wurzel des Templates. Alternativ lege ich vor der Schleife eine Variable an.

DAS NIMMST DU MIT

- `range` und `with` ändern den Punkt.
- `$` ist immer die Wurzel.
- `:=` legt an, `=` ändert.

13

define, include, _helpers.tpl

Bausteine statt Kopien.

Einen Baustein definieren

templates/_helpers.tpl

```
{{- define "web.fullname" -}}
{{- printf "%s-%s" .Release.Name .Chart.Name
    | trunc 63 | trimSuffix "-" -}}
{{- end }}

{{- define "web.selectorLabels" -}}
app.kubernetes.io/name: {{ .Chart.Name }}
app.kubernetes.io/instance: {{ .Release.Name }}
{{- end }}

{{- define "web.labels" -}}
{{ include "web.selectorLabels" . }}
app.kubernetes.io/version: {{ .Chart.AppVersion | quote }}
app.kubernetes.io/managed-by: {{ .Release.Service }}
helm.sh/chart: {{ .Chart.Name }}-{{ .Chart.Version }}
{{- end }}
```

`trunc 63` kürzt den Namen, denn viele Kubernetes-Namen dürfen nur 63 Zeichen lang sein.

Einen Baustein nutzen

```
metadata:
  name: {{ include "web.fullname" . }}
  labels:
    {{- include "web.labels" . | nindent 4 }}
spec:
  selector:
    matchLabels:
      {{- include "web.selectorLabels" . | nindent 6 }}
```

Der Punkt am Ende gibt den Kontext an den Baustein weiter. Vergisst du ihn, hat der Baustein nichts, womit er arbeiten kann.

`include` statt `template`

`template` kann seine Ausgabe nicht in eine Pipe geben. `include` kann das. Darum nimmst du praktisch immer `include`.

> **DAS BILD IM KOPF**
>
> *`define` ist das Rezept im Ordner, `include` ist das Kochen. Der Punkt am Ende sind die Zutaten, die du mitgibst.*

> **STOLPERFALLE**
>
> Benannte Templates sind **global**. Zwei Charts mit dem Baustein `fullname` überschreiben sich gegenseitig. Stelle darum immer den Chart-Namen voran: `web.fullname`.

> **STOLPERFALLE**
>
> Selektor-Labels dürfen sich nie ändern, denn `selector` eines Deployments ist unveränderlich. Eine Version in `matchLabels` lässt das nächste Upgrade scheitern. Darum trennt der Code oben `selectorLabels` und `labels`.

IM INTERVIEW

Warum `include` statt `template`?

`include` gibt einen String zurück, den du weiterverarbeiten kannst, etwa mit `nindent`. `template` schreibt direkt in die Ausgabe. Das verhindert die Einrückung.

DAS NIMMST DU MIT

- `define` im `_helpers.tpl`, `include` im Template.
- Der Punkt gibt den Kontext weiter.
- Namen mit Chart-Präfix.
- Selektor-Labels bleiben stabil.

14

Sprig: die wichtigsten Funktionen

Rund zwanzig Funktionen decken fast alles ab.

Funktion	Wirkung
`default "x" .V`	Ersatzwert bei leerem Wert.
`coalesce .A .B "x"`	Erster nicht leerer Wert.
`quote`, `squote`	Anführungszeichen.
`upper`, `lower`, `title`	Groß- und Kleinschreibung.
`trim`, `trimSuffix`, `trimPrefix`	Abschneiden.
`trunc 63`	Kürzen.
`replace "a" "b"`	Ersetzen.
`printf "%s-%d" .A .B`	Formatieren.
`toYaml`, `toJson`, `fromYaml`	Umwandeln.
`b64enc`, `b64dec`	Base64 für Secrets.
`sha256sum`	Prüfsumme.
`list`, `dict`	Liste und Map bauen.
`hasKey`, `get`, `dig`	In Maps schauen.
`merge`, `mergeOverwrite`	Maps zusammenführen.
`ternary "a" "b" .Bed`	Kurzform für wenn, dann, sonst.
`semverCompare ">=1.30" .V`	Versionen vergleichen.

Das Muster für den Image-Tag

```
image: "{{ .Values.image.repository }}:{{ .Values.image.tag
  | default .Chart.AppVersion }}"
```

Das Muster für neue Pods bei neuer Konfiguration

Ändert sich nur eine ConfigMap, starten die Pods nicht neu. Eine Prüfsumme in den Pod-Annotations löst das:

```
template:
  metadata:
```

```
    annotations:
      checksum/config: {{ include (print $.Template.BasePath
        "/configmap.yaml") . | sha256sum }}
```

Ändert sich die ConfigMap, ändert sich die Prüfsumme, also auch das Pod-Template. Kubernetes rollt neu aus.

STOLPERFALLE

`randAlphaNum` erzeugt bei jedem Rendern einen neuen Wert. Ein zufälliges Passwort in einem Secret ändert sich also bei jedem Upgrade. Lies den alten Wert mit `lookup`, siehe Kapitel 15.

MERKE

Helm entfernt aus Sicherheitsgründen ein paar Sprig-Funktionen, darunter `env` und `expandenv`. Ein Chart soll nicht von der Umgebung des Rechners abhängen, auf dem er gerendert wird.

IM INTERVIEW

Wie startest du Pods neu, wenn sich eine ConfigMap ändert?

Mit einer Prüfsummen-Annotation im Pod-Template. Ändert sich die ConfigMap, ändert sich der Hash und damit das Pod-Template. Das löst ein Rolling Update aus.

DAS NIMMST DU MIT

- `default`, `quote`, `toYaml`, `printf`, `trunc`.
- Prüfsumme im Pod-Template für Config-Rollouts.
- Zufall im Template führt zu Änderungen bei jedem Upgrade.

15

tpl, required, lookup

Drei Funktionen mit Tiefgang.

required: früh und laut scheitern

```
host: {{ required "ingress.host fehlt!" .Values.ingress.host }}
```

Fehlt der Wert, bricht das Rendern mit deiner Meldung ab. Das ist besser als ein leeres Feld, das erst zur Laufzeit auffällt. Für eigene Prüfungen gibt es `fail`:

```
{{- if and .Values.ha (lt (int .Values.replicas) 2) }}
{{- fail "ha braucht mindestens 2 Replikate" }}
{{- end }}
```

tpl: Werte als Template

Normalerweise ist ein Wert nur Text. `tpl` rendert ihn als Template. Damit dürfen Nutzer Ausdrücke in Werten verwenden.

values.yaml

```
hostname: "{{ .Release.Name }}.example.com"
```

```
host: {{ tpl .Values.hostname . }}
```

Das braucht als zweites Argument den Kontext. Ohne `tpl` stünden die Klammern wörtlich im Manifest.

lookup: im Cluster nachsehen

```
{{- $alt := lookup "v1" "Secret" .Release.Namespace "web-db" }}
{{- $pw := randAlphaNum 24 | b64enc }}
{{- if $alt }}
  {{- $pw = index $alt.data "password" }}
{{- end }}
apiVersion: v1
kind: Secret
metadata:
```

```
  name: web-db
data:
  password: {{ $pw }}
```

So bleibt ein einmal erzeugtes Passwort bei Upgrades gleich.

Argument	Bedeutung
1	`apiVersion`, etwa `v1`.
2	`kind`, etwa `Secret`.
3	Namespace, leer für clusterweit.
4	Name, leer für eine Liste.

STOLPERFALLE

`lookup` liefert bei `helm template` und `--dry-run=client` ein leeres Ergebnis, weil Helm den Cluster nicht fragt. Auch Argo CD rendert ohne Lookup. Das Template muss mit leerem Ergebnis funktionieren. `--dry-run=server` fragt den Cluster.

IM INTERVIEW

Was macht `lookup`, und wo sind die Grenzen?

Es liest beim Rendern Objekte aus dem Cluster. Damit behalte ich etwa bestehende Passwörter. Bei `helm template` ist es leer, und die Ausgabe hängt plötzlich vom Clusterzustand ab. Darum nutze ich es sparsam.

DAS NIMMST DU MIT

- `required` für Pflichtwerte, `fail` für Regeln.
- `tpl` rendert Werte als Template.
- `lookup` ist bei `helm template` leer.

16

Dateien einlesen

Konfigurationen als echte Dateien im Chart.

Lange Konfigurationen gehören nicht in YAML-Strings. Lege sie in einen Ordner neben `templates/`.

```
web/
├── config/
│   ├── app.conf
│   └── logging.conf
└── templates/
    └── configmap.yaml
```

templates/configmap.yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ include "web.fullname" . }}
data:
  {{- (.Files.Glob "config/*").AsConfig | nindent 2 }}
```

Aufruf	Wirkung
`.Files.Get "a.conf"`	Inhalt einer Datei als String.
`.Files.Glob "config/*"`	Mehrere Dateien nach Muster.
`.AsConfig`	Formt das Ergebnis für eine ConfigMap.
`.AsSecrets`	Formt es Base64-kodiert für ein Secret.
`.Files.Lines "a.txt"`	Zeile für Zeile, gut für `range`.

Eine einzelne Datei mit Werten darin:

```yaml
data:
  app.conf: |-
    {{- tpl (.Files.Get "config/app.conf") . | nindent 4 }}
```

STOLPERFALLE

Dateien in `templates/` sind für `.Files` nicht sichtbar. Dasselbe gilt für alles, was `.helmignore` ausschließt. Liegt die Datei am falschen Ort, bekommst du still einen leeren String.

IM INTERVIEW

Wie bringst du eine große Konfigurationsdatei in einen Chart?

Ich lege sie in einen Ordner neben `templates/` und lade sie mit `.Files.Get` oder `.Files.Glob` in eine ConfigMap. Braucht sie Werte, rendere ich sie zusätzlich mit `tpl`.

DAS NIMMST DU MIT

- `.Files.Get`, `.Files.Glob`, `.AsConfig`.
- Dateien in `templates/` sind nicht lesbar.
- `tpl` plus `.Files.Get` ergibt Templates in Dateien.

TEIL IV

Fortgeschrittene Charts

Schema, Subcharts, Library Charts, Hooks, Tests und CRDs.

17

values.schema.json

Ein Vertrag für die Werte deines Charts.

Tippfehler in Werten schlagen sonst still fehl. `replicas` statt `replicaCount` tut einfach nichts. Eine JSON-Schema-Datei neben `values.yaml` prüft die Werte.

values.schema.json

```
{
  "$schema": "https://json-schema.org/draft-07/schema#",
  "type": "object",
  "required": ["image"],
  "additionalProperties": false,
  "properties": {
    "replicaCount": { "type": "integer", "minimum": 1 },
    "image": {
      "type": "object",
      "required": ["repository"],
      "properties": {
        "repository": { "type": "string" },
        "tag": { "type": "string" }
      }
    }
  }
}
```

Helm prüft das Schema bei `install`, `upgrade`, `lint` und `template`.

Regel	Fängt
`type`	String statt Zahl.
`required`	Fehlende Pflichtwerte.
`enum`	Werte außerhalb einer Liste.
`minimum`	Unsinnige Zahlen.
`additionalProperties: false`	Tippfehler in Schlüsselnamen.

> **DAS BILD IM KOPF**
>
> *Das Schema ist der Türsteher. `required` im Template merkt erst drinnen, dass etwas fehlt. Das Schema prüft schon am Eingang, und zwar alles auf einmal.*

STOLPERFALLE

`additionalProperties: false` auf oberster Ebene kann Werte für Subcharts ablehnen. Bei einem Chart mit Subcharts musst du deren Schlüssel und `global` im Schema erlauben.

```
helm lint ./web --set replicaCount=abc
```

Eselsbrücke: Gezielt falsche Werte setzen und sehen, ob das Schema sie fängt.

IM INTERVIEW

Wozu dient `values.schema.json`?

Es prüft Werte gegen ein JSON Schema, bevor Helm rendert. So fallen falsche Typen, fehlende Werte und Tippfehler sofort auf, mit klarer Meldung. Zudem dokumentiert es die Schnittstelle des Charts.

DAS NIMMST DU MIT

- Schema prüft bei `install`, `upgrade`, `lint`, `template`.
- `additionalProperties: false` fängt Tippfehler.
- Schema ist Dokumentation, die nicht veraltet.

18

Dependencies und Subcharts

Charts bauen auf Charts auf.

Abhängigkeiten deklarieren

Chart.yaml (Auszug)

```yaml
dependencies:
  - name: postgres
    version: "~15.2.0"
    repository: oci://registry.example.com/charts
    condition: postgres.enabled
  - name: gemeinsam
    version: "1.x.x"
    repository: file://../gemeinsam
    alias: basis
```

Feld	Wirkung
`condition`	Wert, der den Subchart ein- oder ausschaltet.
`tags`	Gruppenschalter für mehrere Subcharts.
`alias`	Anderer Name, auch um denselben Chart doppelt zu nutzen.
`import-values`	Holt Werte des Kindes in den Eltern-Chart.

```
helm dependency update ./web
```

Eselsbrücke: Löse Versionen auf, schreibe `Chart.lock`, lade nach `charts/`.

```
helm dependency build ./web
```

Eselsbrücke: Baue `charts/` exakt nach `Chart.lock`. Das ist der Befehl fürs CI.

```
helm dependency list ./web
```

Eselsbrücke: Was ist deklariert, was ist da?

DAS BILD IM KOPF

`Chart.yaml` ist der Einkaufszettel: „Milch, irgendeine 1,5 %“. `Chart.lock` ist der Kassenbon: genau diese Marke, genau diese Packung. Das CI kauft nach Kassenbon.

Werte an Subcharts geben

Werte des Kindes stehen unter seinem Namen:

`values.yaml`

```yaml
postgres:
  enabled: true
  storage: 20Gi

global:
  registry: registry.example.com
```

Das Kind sieht nur seinen eigenen Abschnitt, und zwar als `.Values.storage`. Nur `global` sehen alle als `.Values.global.registry`.

STOLPERFALLE

Werte in `values.yaml` sind statischer Text. Ein Ausdruck wie der Release-Name wird dort nicht gerendert. Braucht das Kind so einen Wert, muss es ihn selbst mit `tpl` rendern.

MERKE

Subcharts landen zusammen mit dem Eltern-Chart in einem Release. Rechne nicht damit, dass Helm wartet, bis die Datenbank bereit ist, bevor die Anwendung startet. Das muss die Anwendung selbst aushalten.

IM INTERVIEW

`dependency update` oder `dependency build`?

`update` löst die Versionsbereiche neu auf und schreibt `Chart.lock`. `build` nimmt die Versionen aus `Chart.lock`. Im CI nehme ich `build`, damit Builds reproduzierbar sind.

IM INTERVIEW

Wie gibst du Werte an einen Subchart?

In den Werten des Eltern-Charts unter dem Namen des Subcharts. Wenn alle etwas brauchen, kommt es unter `global`.

DAS NIMMST DU MIT

- `Chart.yaml` ist der Wunsch, `Chart.lock` der Beschluss.
- Werte stehen unter dem Subchart-Namen.
- `global` sehen alle.
- `condition` schaltet Subcharts.

19

Library Charts

Geteilter Code ohne eigene Objekte.

Bei 30 Microservices hast du 30 fast gleiche Charts. Ein **Library Chart** enthält nur Bausteine. Er erzeugt keine Objekte und ist nicht installierbar.

`gemeinsam/Chart.yaml`

```yaml
apiVersion: v2
name: gemeinsam
type: library
version: 1.0.0
```

`gemeinsam/templates/_deployment.tpl`

```
{{- define "gemeinsam.deployment" -}}
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}
spec:
  replicas: {{ .Values.replicaCount }}
  # ...
{{- end }}
```

Ein Service bindet die Bibliothek als Abhängigkeit ein und nutzt sie mit einer Zeile:

`templates/deployment.yaml`

```
{{ include "gemeinsam.deployment" . }}
```

> **DAS BILD IM KOPF**
>
> *Ein Application Chart ist ein fertiges Möbelstück. Ein Library Chart ist die Schachtel mit Standard-Schrauben, aus der sich alle Möbel bedienen.*

	Application	Library
`type`	`application`	`library`
Installierbar	Ja	Nein
Erzeugt Objekte	Ja	Nur über `include` im Nutzer
Zweck	Eine Anwendung ausliefern	Muster teilen

STOLPERFALLE

Eine Änderung in der Bibliothek betrifft alle Nutzer. Versioniere sie streng mit SemVer und aktualisiere Services bewusst, nicht alle auf einmal.

IM INTERVIEW

Wann lohnt sich ein Library Chart?

Wenn viele Charts die gleichen Muster teilen, etwa Labels, Deployments oder Security-Kontexte. Änderungen zentral und versioniert, statt in 30 Kopien. Für zwei Charts lohnt sich der Aufwand selten.

DAS NIMMST DU MIT

- `type: library`, nur `define`.
- Einbinden über `dependencies` und `include`.
- Streng versionieren.

20

Hooks

Eingriffe an bestimmten Punkten des Lebenszyklus.

Ein Hook ist ein normales Objekt mit einer Annotation. Helm führt es zu einem bestimmten Zeitpunkt aus. Typisch sind Datenbank-Migrationen.

`templates/migration.yaml`

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: {{ include "web.fullname" . }}-migration
  annotations:
    "helm.sh/hook": pre-upgrade,pre-install
    "helm.sh/hook-weight": "-5"
    "helm.sh/hook-delete-policy": before-hook-creation
spec:
  backoffLimit: 1
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: migrate
          image: {{ .Values.migration.image | quote }}
          command: ["./migrate"]
```

Die Zeitpunkte

Hook	Läuft
`pre-install`, `post-install`	Vor und nach der Installation.
`pre-upgrade`, `post-upgrade`	Vor und nach einem Upgrade.
`pre-rollback`, `post-rollback`	Vor und nach einem Rollback.
`pre-delete`, `post-delete`	Vor und nach dem Löschen.
`test`	Beim Aufruf von `helm test`.

Reihenfolge und Aufräumen

- `hook-weight`: Hooks laufen aufsteigend. Der Wert ist ein String.
- `hook-delete-policy`: Wann das Objekt verschwindet.

Policy	Bedeutung
`before-hook-creation`	Altes Objekt vor dem nächsten Lauf löschen. Standard.
`hook-succeeded`	Nach Erfolg löschen.
`hook-failed`	Nach Fehlschlag löschen.

DAS BILD IM KOPF

Hooks sind wie die Crew vor und nach dem Konzert: Sie bauen auf und räumen ab, stehen aber nicht im Programmheft. Sie gehören nicht zum Release.

STOLPERFALLE

Hooks gehören nicht zum Release. Helm verwaltet sie nicht mit: `helm uninstall` löscht sie nicht, und der 3-Way-Merge sieht sie nicht. Ohne `hook-delete-policy` bleiben alte Jobs liegen.

STOLPERFALLE

Ein `pre-install`-Job, der ein Secret aus dem Chart braucht, findet es nicht, denn normale Objekte gibt es zu diesem Zeitpunkt noch nicht. Mach das Secret selbst zum Hook mit niedrigerem Gewicht.

IM INTERVIEW

Wie führst du Datenbank-Migrationen mit Helm aus?

Als `pre-upgrade`-Job mit `backoffLimit` und klarer Delete-Policy. Scheitert er, scheitert das Upgrade, bevor neue Pods starten. Wichtig: Die Migration muss abwärtskompatibel sein, damit ein Rollback klappt.

IM INTERVIEW

Sind Hooks Teil des Releases?

Nein. Helm führt sie aus und wartet, verwaltet sie aber nicht wie Release-Objekte. Sie brauchen eine eigene Delete-Policy.

DAS NIMMST DU MIT

- Annotation `helm.sh/hook` macht ein Objekt zum Hook.
- Gewicht steuert die Reihenfolge.
- Hooks stehen außerhalb des Releases.
- `helm install --no-hooks` überspringt sie.

21

Chart-Tests

Läuft die Anwendung wirklich, nachdem Helm fertig ist?

Ein Chart-Test ist ein Pod mit einer Hook-Annotation. Endet er mit Exit-Code 0, ist der Test bestanden.

`templates/tests/verbindung.yaml`

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: {{ include "web.fullname" . }}-test
  annotations:
    "helm.sh/hook": test
spec:
  restartPolicy: Never
  containers:
    - name: curl
      image: curlimages/curl:8.11.0
      args:
        - --fail
        - >-
          http://{{ include "web.fullname" . }}:{{
          .Values.service.port }}/
```

```
helm test web -n demo --logs
```

Eselsbrücke: Starte die Tests im Cluster und zeig danach ihre Logs.

Chart-Tests prüfen den **laufenden** Release. Für das Rendern sind Unit-Tests da, siehe Kapitel 31.

Werkzeug	Prüft	Cluster?
`helm lint`	Struktur und Schema.	Nein
`helm-unittest`	Gerenderte Ausgabe.	Nein
`helm test`	Die laufende Anwendung.	Ja

IM INTERVIEW

Wie testest du einen Helm-Chart?

In Schichten: `helm lint`, Unit-Tests mit `helm-unittest`, Schema-Prüfung der Manifeste, dann in einem Wegwerf-Cluster installieren und `helm test` starten.

DAS NIMMST DU MIT

- Test ist ein Pod mit `"helm.sh/hook": test`.
- Exit-Code 0 heißt bestanden.
- `helm test` braucht einen laufenden Release.

22

CRDs

Der wunde Punkt von Helm.

Dateien im Ordner `crds/` behandelt Helm besonders:

Regel	Folge
Keine Templates	Kein Zugriff auf Werte.
Vor allem anderen installiert	Objekte im Chart dürfen die CRD direkt nutzen.
Nur beim ersten Install	`helm upgrade` ändert sie nie.
Nie gelöscht	`helm uninstall` lässt sie stehen.

Warum diese Vorsicht? CRDs gelten clusterweit. Wird eine CRD gelöscht, verschwinden alle ihre Custom Resources in allen Namespaces. Ein Upgrade eines Charts kann also fremde Daten treffen.

> **DAS BILD IM KOPF**
>
> *Eine CRD ist ein Gesetz für den ganzen Staat. Ein Chart ist ein einzelner Bürger. Deshalb darf ein Bürger das Gesetz einmal einbringen, aber nicht nebenbei ändern oder abschaffen.*

Wege für Updates

Weg	Abwägung
Eigener CRD-Chart	CRDs in `templates/` eines getrennten Charts. Sauber und getrennt versioniert.
Direkt mit `kubectl apply`	Einfach, aber ein extra Schritt im Prozess.
Operator verwaltet sie	Manche Projekte bringen das selbst mit.
`crds/` plus Handarbeit	Neue CRD-Versionen musst du selbst anwenden.

```
helm show crds ./web | kubectl apply --server-side -f -
```

Eselsbrücke: Hol die CRDs aus dem Chart und wende sie selbst an. Das ist der Handweg.

STOLPERFALLE

CRD in `templates/` und im selben Chart Objekte dieser CRD geht schief, denn Helm prüft die Objekte vor dem Anlegen und kennt die Art noch nicht: `no matches for kind`. Hier hilft ein getrennter Chart.

STOLPERFALLE

Liegen CRDs in `templates/`, löscht `helm uninstall` sie mit. Das reißt alle Custom Resources mit. Schütze sie mit `"helm.sh/resource-policy": keep`.

IM INTERVIEW

Warum aktualisiert Helm CRDs im Ordner `crds/` nicht?

CRDs gelten clusterweit, und ihre Änderung oder Löschung trifft alle Custom Resources. Darum installiert Helm sie nur einmal. Updates mache ich bewusst, oft über einen eigenen CRD-Chart.

DAS NIMMST DU MIT

- `crds/`: einmal installiert, nie geändert, nie gelöscht.
- Eigener CRD-Chart für saubere Updates.
- `resource-policy: keep` schützt beim Löschen.

TEIL V

Betrieb

Upgrades, Release-Speicher, Apply, Helm 4, Registries, Signaturen und Secrets.

23

install, upgrade, rollback

Der eine Befehl für jedes Deployment.

Der Standardbefehl

```
helm upgrade --install web \
  oci://registry.example.com/charts/web --version 1.4.2 \
  --namespace prod --create-namespace \
  -f values/prod.yaml \
  --wait --timeout 5m \
  --rollback-on-failure
```

`upgrade --install` ist **idempotent**. Gibt es den Release, wird er aktualisiert. Sonst wird er installiert. Darum läuft dieselbe Zeile beim ersten und beim hundertsten Mal.

> **DAS BILD IM KOPF**
>
> *`upgrade --install` ist wie der Lichtschalter mit Fernbedienung: Du drückst „an“, egal ob das Licht schon brennt. Das Ergebnis ist immer dasselbe.*

Warten

Ohne `--wait` meldet Helm Erfolg, sobald die API die Objekte angenommen hat. Die Pods können dann noch abstürzen. Standard ist `hookOnly`, also warten nur auf Hooks.

Strategie	Wirkung
`--wait`	Kurz für `--wait=watcher`. Beobachtet den Status mit kstatus.
`--wait=hookOnly`	Standard: Wartet nur auf Hooks.
`--wait=legacy`	Das Warteverhalten von Helm 3.
`--wait-for-jobs`	Wartet zusätzlich, bis Jobs fertig sind.

`--timeout` begrenzt die Wartezeit. Standard sind fünf Minuten.

> **STOLPERFALLE**
>
> Ohne `readinessProbe` ist ein Pod schon bereit, wenn der Prozess startet. Dann nützt `--wait` wenig. Helm ist nur so gut wie deine Probes.

Wenn es schiefgeht

`--rollback-on-failure` rollt ein gescheitertes Upgrade automatisch zurück. Bei einer Erstinstallation entfernt es den Release wieder. Das Flag schaltet `--wait` mit ein. In Helm 3 hieß es `--atomic`.

Rollback

```
helm history web -n prod
```

Eselsbrücke: Welche Revision war die letzte gute?

```
helm rollback web 4 -n prod --wait
```

Eselsbrücke: Zurück auf den Stand von Revision 4, als neue Revision.

Ein Rollback stellt nur Manifeste wieder her.

> **STOLPERFALLE**
>
> Ein Rollback dreht keine Datenbank-Migration zurück und ändert keine Daten in PVCs. Bei Änderungen am Schema muss die alte Version mit dem neuen Schema laufen können. Plane das vor dem Upgrade.

Vorher schauen

```
helm upgrade web ./web -n prod -f prod.yaml --dry-run=server
```

Eselsbrücke: Simuliere gegen den Cluster, ohne etwas zu ändern.

Modus	Cluster?
`--dry-run=client`	Kein Kontakt, wie `helm template`.
`--dry-run=server`	Fragt den Cluster, auch für `lookup`.

```
helm diff upgrade web ./web -n prod -f prod.yaml
```

Eselsbrücke: Das Plugin `helm-diff` zeigt wie `git diff`, was sich ändern würde.

IM INTERVIEW

Wie sieht ein sicheres Upgrade in Produktion aus?

Ein fester Chart-Version-Pin, Werte aus Dateien im Git, vorher ein Diff, `upgrade --install` mit `--wait`, Timeout und `--rollback-on-failure`. Danach `helm test` und ein Blick aufs Monitoring.

IM INTERVIEW

Was stellt `helm rollback` wieder her, und was nicht?

Die Manifeste einer früheren Revision. Nicht wiederhergestellt werden Daten, PVCs oder Datenbank-Schemata. Das Rollback ist selbst eine neue Revision.

DAS NIMMST DU MIT

- `helm upgrade --install` ist idempotent.
- `--wait` und `--rollback-on-failure` gehören zusammen.
- `--dry-run=server` und `helm diff` vor Produktion.
- Rollback dreht Daten nicht zurück.

24

Wo Helm seine Releases speichert

Das Gedächtnis von Helm ist ein Secret im Namespace.

Helm hat keinen Server. Sein Gedächtnis liegt im Cluster, und zwar im Namespace des Releases.

```
kubectl get secrets -n demo -l owner=helm
```

Eselsbrücke: Das Label zeigt alle Release-Secrets.

```
NAME                         TYPE                 DATA  AGE
sh.helm.release.v1.web.v1  helm.sh/release.v1   1     2d
sh.helm.release.v1.web.v2  helm.sh/release.v1   1     1h
```

Pro Revision gibt es ein Secret. Der Name verrät Release und Revision. Darin steckt alles: Chart, Werte, gerendertes Manifest und Status.

Hineinschauen

```
kubectl get secret sh.helm.release.v1.web.v2 -n demo \
  -o jsonpath='{.data.release}' \
  | base64 -d | base64 -d | gzip -d
```

Zweimal `base64`, denn Kubernetes kodiert das schon von Helm kodierte Gzip noch einmal.

Release-Status

Status	Bedeutung
`deployed`	Die aktuelle, aktive Revision.
`superseded`	Eine ältere, abgelöste Revision.
`failed`	Die Revision ist gescheitert.
`pending-install`, `pending-upgrade`, `pending-rollback`	Eine Aktion läuft, oder sie hängt.
`uninstalling`, `uninstalled`	Wird entfernt, oder ist mit `--keep-history` entfernt.

Grenzen

- Ein Secret darf maximal 1 MiB groß sein. Sehr große Charts stoßen daran.
- `--history-max` begrenzt die Revisionen, Standard sind 10. Ältere werden gelöscht.
- `HELM_DRIVER` wählt den Speicher: `secret` (Standard), `configmap`, `memory` oder `sql`.

DAS BILD IM KOPF

Das Secret ist das Logbuch an Bord des Schiffs, nicht im Hafenbüro. Wer das Schiff (den Namespace) kennt, findet die ganze Reise darin.

STOLPERFALLE

Wer Secrets im Namespace lesen darf, kann auch alle Werte des Releases lesen, mitsamt Passwörtern aus `--set`. Das ist nur komprimiert, nicht verschlüsselt. Ein Löschen des Namespaces entfernt auch die Historie.

IM INTERVIEW

Wo speichert Helm seinen Zustand?

Als Secret vom Typ `helm.sh/release.v1` im Namespace des Releases, eines pro Revision. Darin stehen Chart, Werte und Manifest. Warum dort? Es gibt keinen Server, und RBAC gilt so automatisch.

DAS NIMMST DU MIT

- Ein Secret pro Revision.
- `helm get` liest diese Secrets.
- Nur komprimiert, nicht verschlüsselt.
- `--history-max` hält die Liste kurz.

25

3-Way-Merge und Server-Side Apply

Wie Helm entscheidet, was es im Cluster ändert.

Das Problem

Im Cluster ändern auch andere Dinge deine Objekte: ein HPA, ein Sidecar-Injector, ein Admin mit `kubectl edit`. Soll Helm das überschreiben?

Client-Side: der 3-Way-Merge

Helm vergleicht drei Stände:

```
alt:  letztes Manifest aus dem Release
live: was jetzt im Cluster steht
neu:  was der Chart jetzt rendert
```

Alt	Live	Neu	Ergebnis
2	2	3	Chart hat geändert: wird 3.
2	5	2	Chart unverändert: Live bleibt 5.
2	5	3	Beide geändert: Chart gewinnt, wird 3.

Das heißt: Felder, die der Chart nicht ändert, bleiben, wie sie im Cluster sind. Helm 2 kannte nur zwei Stände und übersah dabei manuelle Änderungen.

> **DAS BILD IM KOPF**
>
> *Git-Merge für den Cluster: gemeinsamer Vorfahre, deine Änderung, ihre Änderung.*

Server-Side Apply

Mit **Server-Side Apply** (SSA) rechnet nicht Helm, sondern der API-Server. Er merkt sich pro Feld einen **Field Manager**, also wem das Feld gehört. Will Helm ein Feld ändern, das einem anderen gehört, meldet die API einen Konflikt statt es still zu überschreiben.

```
helm upgrade web ./web -n prod --force-conflicts
```

Eselsbrücke: Bewusste Entscheidung: Bei einem Konflikt übernimmt Helm das Feld trotzdem.

Was gilt wann?

Situation	Methode in Helm 4
Neue Installation	Server-Side Apply.
Upgrade, Rollback	Wie beim Release bisher (`--server-side=auto`).
Release aus Helm 3	Bleibt Client-Side.
Ausdrücklich gewählt	`--server-side=true` oder `false`.

Drift

Drift heißt: Der Cluster weicht vom Chart ab. Helm erkennt das nicht von selbst. Nur `helm diff` oder ein GitOps-Werkzeug macht Drift sichtbar.

> **STOLPERFALLE**
>
> `replicas` fest im Chart und zugleich ein HPA: Beide streiten um dasselbe Feld. Lass das Feld im Template weg, wenn ein HPA aktiv ist. Mit SSA zeigt sich so ein Streit oft als Konflikt.

IM INTERVIEW

Was ist der 3-Way-Merge?

Helm vergleicht das letzte Release-Manifest, den Live-Stand und das neue Manifest. So bleiben fremde Änderungen an Feldern erhalten, die der Chart nicht anfasst, während Chart-Änderungen ankommen.

IM INTERVIEW

Was ändert Server-Side Apply?

Der API-Server führt das Zusammenführen aus und verfolgt Eigentum pro Feld. Konflikte werden sichtbar, statt still überschrieben zu werden. Helm 4 nutzt das für neue Installationen, bestehende Releases behalten ihre Methode.

DAS NIMMST DU MIT

- Client-Side: alt, live, neu.
- SSA: Field Manager und Konflikte.
- Neue Installationen in Helm 4: SSA.
- Drift erkennt Helm nicht allein.

26

Was Helm 4 ändert

Der Wechsel von Helm 3, kurz und vollständig genug fürs Gespräch.

Helm 4 erschien im November 2025. Bestehende `v2`-Charts laufen weiter. Es ändert vor allem die Arbeitsweise drumherum.

Umbenannte Flags

Helm 3	Helm 4
`--atomic`	`--rollback-on-failure`
`--force`	`--force-replace`

Die alten Namen funktionieren noch, geben aber eine Warnung aus.

Brüche, die du prüfen musst

- **Post-Renderer** sind Plugins. Ein Pfad zu einem Programm reicht nicht mehr.
- `helm registry login` nimmt nur noch den Domainnamen, keine ganze URL.
- **Server-Side Apply** ist bei neuen Installationen der Standard.
- **Skripte**, die die Ausgabe von Helm parsen, sollten neu geprüft werden.

Neues

Thema	Neuerung
Warten	`--wait` nutzt kstatus und zeigt genauer, woran es hängt.
Plugins	Neues System mit optionaler WebAssembly-Laufzeit.
Plugin-Typen	CLI, Getter und Post-Renderer.
Template-Funktionen	Plugins können eigene Funktionen hinzufügen.
OCI	Installation per Digest (`@sha256:...`).
Werte	Werte lassen sich auf mehrere YAML-Dokumente verteilen.
Cache	Inhaltsbasierter Cache für Charts.
SDK	Eine stabile Go-API.

> **MERKE**
>
> Helm 3 bekommt seit der Sonderrelease vom 9. September 2026 nur noch Sicherheitsupdates, und zwar bis zum 10. Februar 2027. Danach ist Schluss. Plane den Wechsel jetzt.

Der Wechsel in der Praxis

1. Pipelines auf die neuen Flag-Namen umstellen.
2. Plugins prüfen. Für `helm-unittest` braucht die Installation unter Helm 4 `--verify=false`.
3. Post-Renderer als Plugin umbauen.
4. In einem Testcluster einen Helm-3-Release mit Helm 4 aktualisieren.
5. Prüfen, wie sich Server-Side Apply auf Felder auswirkt, die andere Werkzeuge verwalten.

IM INTERVIEW

Was ist der wichtigste Unterschied zwischen Helm 3 und Helm 4?

Server-Side Apply als Standard für neue Installationen, Warten mit kstatus und ein neues Plugin-System mit WebAssembly. Dazu umbenannte Flags wie `--rollback-on-failure`. Bestehende Charts laufen weiter.

IM INTERVIEW

Laufen Helm-3-Releases unter Helm 4 weiter?

Ja. Helm 4 erkennt sie, und Upgrades behalten die bisherige Apply-Methode. Sie bleiben also bei Client-Side Apply, solange du nicht ausdrücklich wechselst.

DAS NIMMST DU MIT

- `--atomic` wurde `--rollback-on-failure`, `--force` wurde `--force-replace`.
- SSA für neue Installationen.
- Post-Renderer sind jetzt Plugins.
- Helm 3 endet am 10. Februar 2027.

27

OCI-Registries

Charts liegen dort, wo auch deine Images liegen.

Eine OCI-Registry speichert nicht nur Images, sondern beliebige Artefakte. Helm nutzt das für Charts. Du brauchst keinen eigenen Chart-Server und keine `index.yaml`.

Veröffentlichen

```
helm registry login registry.example.com
```

Eselsbrücke: Nur die Domain, ohne `oci://` und ohne Pfad.

```
helm package ./web
```

Eselsbrücke: Erzeugt `web-1.4.2.tgz`.

```
helm push web-1.4.2.tgz oci://registry.example.com/charts
```

Eselsbrücke: Name und Tag stammen aus dem Paket. Du gibst nur den Ordner an.

Nutzen

```
helm show chart oci://registry.example.com/charts/web --ver-
sion 1.4.2
```

Eselsbrücke: Metadaten lesen, ohne zu installieren.

```
helm install web oci://registry.example.com/charts/web --
version 1.4.2
```

Eselsbrücke: Der Pfad ersetzt `repo/chart`. `helm repo add` ist nicht nötig.

Per Digest pinnen

Ein Tag kann überschrieben werden, ein Digest nicht.

```
helm install web \
  oci://registry.example.com/charts/web@sha256:abc123...
```

Stimmt der Digest nicht, installiert Helm nicht.

OCI oder HTTP-Repository?

	OCI	HTTP-Repository
Infrastruktur	Vorhandene Registry	Webserver und `index.yaml`
Rechte	Wie bei Images	Eigene Lösung
`helm repo add`	Nicht nötig	Nötig
`helm search repo`	Geht nicht	Geht
Auflösung per Digest	Ja	Nein

DAS BILD IM KOPF

Früher hatten Charts ein eigenes Lager mit eigenem Katalog. Jetzt liegen sie im selben Regal wie die Images, mit derselben Inventur und denselben Schlüsseln.

STOLPERFALLE

`+` ist in SemVer erlaubt, aber nicht in OCI-Tags. Helm ersetzt es im Tag durch `_`. Ein Chart `1.0.0+build1` erscheint also als `1.0.0_build1`.

IM INTERVIEW

Warum Charts in einer OCI-Registry?

Eine einzige Infrastruktur für Images und Charts, mit denselben Rechten, Scans und Replikationen. Ich pinne per Digest und brauche keinen Index-Server mehr.

DAS NIMMST DU MIT

- `helm registry login`, `package`, `push`.
- `oci://` ersetzt `helm repo add`.
- Digest ist unveränderlich, ein Tag nicht.

28

Signieren und Provenance

Woher weißt du, dass der Chart echt ist?

Ein Chart führt beliebigen Code in deinem Cluster aus. Du willst also wissen, von wem er stammt und ob ihn jemand verändert hat.

Der Helm-eigene Weg: PGP

```
helm package ./web --sign --key 'Team Plattform' --keyring ring.gpg
```

Eselsbrücke: `--key` ist ein Teil des Namens im Schlüssel, kein Fingerabdruck.

Das erzeugt neben dem Paket eine Datei `web-1.4.2.tgz.prov` mit Prüfsumme und Signatur.

```
helm verify web-1.4.2.tgz --keyring public.gpg
```

Eselsbrücke: Echtheit prüfen, lokal.

```
helm install web ./web-1.4.2.tgz --verify --keyring public.gpg
```

Eselsbrücke: Installation nur, wenn die Prüfung klappt.

Der moderne Weg: cosign

In OCI-Registries lässt sich ein Chart wie ein Image mit `cosign` signieren. Das Sigstore-Projekt erlaubt auch schlüssellose Signaturen über die Identität im CI.

```
cosign sign \
  registry.example.com/charts/web@sha256:abc123...
```

DAS BILD IM KOPF

Der Digest beweist, dass das Paket unversehrt ist. Die Signatur beweist, wer es gepackt hat. Du brauchst für echtes Vertrauen beides.

STOLPERFALLE

Zum Signieren braucht Helm den privaten Schlüssel in einer Keyring-Datei. Liegt er nur im Schlüsselspeicher von GnuPG 2, exportierst du ihn zuerst: `gpg --export-secret-keys > ~/.gnupg/secring.gpg`. Zum Prüfen liest Helm ab 4.3.0 auch die Keybox von GnuPG direkt.

IM INTERVIEW

Wie sicherst du die Lieferkette von Charts ab?

Charts aus einer eigenen Registry, per Digest gepinnt, im CI signiert und vor der Installation geprüft. Dazu Scans der gerenderten Manifeste. Fremde Charts spiegle ich und lese sie vorher.

DAS NIMMST DU MIT

- `.prov` plus `helm verify` für PGP.
- `cosign` für OCI.
- Digest für Integrität, Signatur für Herkunft.

29

Secrets

Was nie in einer values.yaml im Git stehen darf.

Es gibt zwei Lecks:

- **Git:** Ein Passwort in `values.yaml` bleibt für immer in der Historie.
- **Release:** Werte aus `--set` stehen im Release-Secret. `helm get values` zeigt sie.

Base64 ist keine Verschlüsselung.

Die Muster

Weg	Idee
`existingSecret`	Chart verweist nur auf ein Secret, das anders entsteht.
External Secrets Operator	Ein Controller holt Daten aus Vault, AWS oder Ähnlichem.
Sealed Secrets	Verschlüsseltes Objekt liegt im Git, nur der Cluster kann es öffnen.
SOPS mit `helm-secrets`	Verschlüsselte Werte-Dateien im Git, beim Deployment entschlüsselt.
CSI Secrets Store	Das Geheimnis wird als Datei in den Pod eingehängt.

Das Muster `existingSecret`

Der Chart sollte immer einen Verweis erlauben:

values.yaml

```yaml
datenbank:
  existingSecret: ""
  passwordKey: password
```

templates/deployment.yaml (Auszug)

```yaml
env:
  - name: DB_PASSWORD
    valueFrom:
      secretKeyRef:
        name: {{ required "datenbank.existingSecret fehlt"
```

```
          .Values.datenbank.existingSecret }}
        key: {{ .Values.datenbank.passwordKey }}
```

External Secrets

Der Chart liefert nur die Anforderung aus, nicht das Geheimnis:

templates/externalsecret.yaml

```yaml
apiVersion: external-secrets.io/v1
kind: ExternalSecret
metadata:
  name: {{ include "web.fullname" . }}-db
spec:
  secretStoreRef:
    name: vault
    kind: ClusterSecretStore
  target:
    name: {{ include "web.fullname" . }}-db
  data:
    - secretKey: password
      remoteRef:
        key: web/prod/db
        property: password
```

DAS BILD IM KOPF

Der Chart trägt keinen Schlüssel bei sich. Er hat nur einen Zettel, auf dem steht, bei welchem Pförtner der Schlüssel liegt.

STOLPERFALLE

Ein per `{{ .Values.pw | b64enc }}` gerendertes Secret ist nicht sicher. Der Wert steht im Git oder in der CI-Variable, im Release-Secret und im Manifest. Und `helm template` im CI-Log zeigt ihn ebenfalls.

IM INTERVIEW

Wie gehst du mit Secrets in Helm um?

Keine Klartext-Werte im Git. Am liebsten holt External Secrets die Daten aus einem Vault, und der Chart verweist nur darauf. Wer ohne Vault arbeitet, nimmt SOPS oder Sealed Secrets.

DAS NIMMST DU MIT

- Werte stehen im Git und im Release-Secret.
- Chart verweist, liefert keine Geheimnisse.
- External Secrets, Sealed Secrets, SOPS.

TEIL VI

Qualität und Auslieferung

Lint, Unit-Tests, CI, GitOps und der Vergleich mit Kustomize.

30

Lint und Rendern

Fehler finden, bevor ein Cluster sie sieht.

```
helm lint ./web --strict
```

Eselsbrücke: **Lint** ist die Fussel-Bürste. `--strict` macht aus Warnungen Fehler.

```
helm lint ./web --with-subcharts -f values/prod.yaml
```

Eselsbrücke: Prüfe auch Subcharts, mit den echten Werten.

> **MERKE**
>
> `helm lint` kennt nur Chart-Regeln. Es prüft nicht, ob das Ergebnis gültiges Kubernetes ist. Ein Tippfehler wie `replica:` statt `replicas:` kommt durch.

Das Ergebnis gegen Kubernetes prüfen

```
helm template web ./web -f values/prod.yaml | kubeconform -strict -summary
```

Eselsbrücke: Prüfe das Ergebnis gegen die Kubernetes-Schemas, ohne Cluster.

```
helm template web ./web | kubectl apply --dry-run=server -f -
```

Eselsbrücke: Frag einen echten API-Server. Das fängt auch Admission-Regeln.

Jede Umgebung rendern

Der Chart ist nur so gut wie die Kombination seiner Werte. Prüfe im CI jede Umgebung:

```
for env in dev staging prod; do
  helm template web ./web -f "values/$env.yaml" \
    | kubeconform -strict -summary
done
```

Sicherheit prüfen

Werkzeuge wie `trivy config`, `kube-linter` oder `checkov` finden Muster wie privilegierte Container oder fehlende Limits im gerenderten YAML.

> **DAS BILD IM KOPF**
>
> *Lint prüft die Rechtschreibung, kubeconform die Grammatik, der Dry Run fragt den Hausmeister, und der Sicherheitsscan fragt, ob die Tür offen steht.*

IM INTERVIEW

Was prüft `helm lint`, und was nicht?

Es prüft Aufbau, Template-Syntax und Schema. Es prüft nicht, ob die Manifeste gültige Kubernetes-Objekte sind. Dafür rendere ich und prüfe mit `kubeconform` oder einem Server-Dry-Run.

DAS NIMMST DU MIT

- `helm lint --strict` ist nur der Anfang.
- `helm template | kubeconform` prüft das Ergebnis.
- Jede Umgebung rendern.

31

helm-unittest

Tests für Templates, ohne Cluster.

Bei vielen Werten und Bedingungen kannst du nicht alles von Hand prüfen. `helm-unittest` rendert deine Templates und prüft das Ergebnis.

```
helm plugin install https://github.com/helm-unittest/helm-unittest.git --verify=false
```

Eselsbrücke: Unter Helm 4 braucht dieses Plugin `--verify=false`.

Tests liegen in `tests/` und enden auf `_test.yaml`.

tests/deployment_test.yaml

```yaml
suite: deployment
templates:
  - deployment.yaml
tests:
  - it: nimmt appVersion als Standard-Tag
    release:
      name: web
    chart:
      appVersion: "2.7.0"
    asserts:
      - isKind:
          of: Deployment
      - equal:
          path: spec.template.spec.containers[0].image
          value: registry.example.com/web:2.7.0

  - it: setzt die Replikate aus den Werten
    set:
      replicaCount: 5
    asserts:
      - equal:
          path: spec.replicas
          value: 5
```

```
helm unittest ./web
```

Eselsbrücke: Führt alle Tests im Chart aus.

Wichtige Assertions

Assertion	Prüft
`equal`	Wert an einem Pfad.
`matchRegex`	Muster an einem Pfad.
`contains`	Element in einer Liste.
`notExists`, `exists`	Ob ein Pfad vorhanden ist.
`hasDocuments`	Anzahl gerenderter Dokumente.
`failedTemplate`	Ob das Rendern scheitert, etwa bei `required`.
`matchSnapshot`	Vergleich mit gespeicherter Ausgabe.

Was du testest: die Bedingungen. Jedes `if`, jedes `required`, und jeder Standardwert verdient einen Test.

> **STOLPERFALLE**
>
> Snapshots bestehen auch dann, wenn das Falsche gespeichert wurde. Wer bei jeder Abweichung nur den Snapshot neu schreibt, testet nichts. Nutze gezielte Assertions für das Wichtige.

IM INTERVIEW

Was testest du in einem Chart mit Unit-Tests?

Verhalten: Bedingungen ein und aus, Standardwerte, Pflichtwerte, die scheitern müssen, und Pfade, die in einer Konfiguration fehlen sollen. Nicht jede YAML-Zeile.

DAS NIMMST DU MIT

- Tests in `tests/*_test.yaml`.
- `set`, `asserts`, `equal`, `failedTemplate`.
- Bedingungen testen, nicht jede Zeile.

32

CI-Pipeline

Vom Commit bis zum signierten Chart.

Die Stufen

1. **Lint:** `helm lint --strict`.
2. **Unit-Tests:** `helm unittest`.
3. **Rendern:** jede Umgebung mit `kubeconform` prüfen.
4. **Scan:** `trivy config` auf das Ergebnis.
5. **Integration:** `ct install` in einem `kind`-Cluster.
6. **Veröffentlichen:** bei einem Tag `package`, `push`, signieren.

Chart Testing

Das offizielle Werkzeug `ct` (**chart-testing**) findet geänderte Charts in einem Monorepo.

```
ct lint --target-branch main
```

Eselsbrücke: Lint für geänderte Charts. Prüft auch, ob die Version erhöht wurde.

```
ct install --target-branch main
```

Eselsbrücke: Installiert sie in einem Cluster und startet `helm test`.

Ein Beispiel

`.github/workflows/chart.yaml (Kern)`

```yaml
on:
  push:
    tags: ["v*"]

jobs:
  release:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v7
      - uses: azure/setup-helm@v5
```

```
      - run: helm dependency build ./web
      - run: helm lint ./web --strict
      - run: |
          helm package ./web \
            --version "${GITHUB_REF_NAME#v}"
      - run: |
          echo "$REG_PASSWORD" | helm registry login \
            registry.example.com -u ci --password-stdin
          helm push web-*.tgz \
            oci://registry.example.com/charts
        env:
          REG_PASSWORD: ${{ secrets.REG_PASSWORD }}
```

MERKE

Die Pipeline pinnt am besten auch die Helm-Version und Actions per Commit-Hash. Dann ändert sich nicht heimlich das Werkzeug, das deine Charts baut.

STOLPERFALLE

Dieselbe Version zweimal in die Registry zu schieben, überschreibt je nach Registry das Paket. Eine veröffentlichte Chart-Version ist unveränderlich. Bei einer Änderung erhöhst du die Version.

IM INTERVIEW

Wie sieht eine CI-Pipeline für Charts aus?

Lint, Unit-Tests, Rendern aller Umgebungen mit Schema-Prüfung, Sicherheitsscan, Installation in `kind` mit `helm test`. Bei einem Tag paketieren, in die OCI-Registry schieben und signieren.

DAS NIMMST DU MIT

- Lint, Test, Rendern, Scan, Integration, Veröffentlichen.
- `ct` für Monorepos.
- Chart-Versionen sind unveränderlich.

33

GitOps: Argo CD und Flux

Git sagt, was laufen soll. Ein Controller sorgt dafür.

Beim GitOps ruft niemand mehr `helm upgrade` auf. Ein Controller im Cluster vergleicht den Sollzustand im Git mit der Wirklichkeit und gleicht sie an.

Argo CD

application.yaml

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: web
  namespace: argocd
spec:
  project: default
  source:
    repoURL: registry.example.com/charts
    chart: web
    targetRevision: 1.4.2
    helm:
      valueFiles:
        - values-prod.yaml
  destination:
    server: https://kubernetes.default.svc
    namespace: prod
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

Argo CD rendert mit `helm template` und wendet das Ergebnis selbst an. Helm verwaltet also keinen Release. Seit Version 3.5 rendert Argo CD nur noch mit Helm 4.

Flux

helmrelease.yaml

```yaml
apiVersion: helm.toolkit.fluxcd.io/v2
kind: HelmRelease
```

```yaml
metadata:
  name: web
  namespace: prod
spec:
  interval: 10m
  chart:
    spec:
      chart: web
      version: 1.4.2
      sourceRef:
        kind: HelmRepository
        name: firma
  values:
    replicaCount: 3
```

Der Helm Controller von Flux führt echte Helm-Aktionen aus. Seit Flux 2.8 nutzt er Helm 4: Neue Releases laufen mit Server-Side Apply, bestehende bleiben bei Client-Side Apply, bis du umstellst. Die Prüfung mit kstatus gilt für alle. Für Produktion empfiehlt Flux OCI-Charts mit `OCIRepository` und `chartRef`.

Der wichtige Unterschied

	Argo CD	Flux
Wie	Rendert und wendet selbst an	Führt Helm-Aktionen aus
Helm 4	Ab Argo CD 3.5	Ab Flux 2.8
`helm list`	Zeigt nichts	Zeigt den Release
Hooks	Werden auf Argo-Hooks abgebildet. Rollback- und Test-Hooks werden ignoriert. Definierst du eigene Argo-Hooks, ignoriert Argo alle Helm-Hooks.	Laufen wie bei Helm
`lookup`	Liefert leer	Funktioniert
Rollback	Über Git oder die Oberfläche	Über Git, oder automatisch, wenn Remediation konfiguriert ist

> **DAS BILD IM KOPF**
>
> *Argo CD liest das Rezept und kocht selbst. Flux gibt das Rezept dem Helm-Koch und überwacht das Ergebnis. Das Gericht ist gleich, aber nur beim zweiten steht ein Helm-Eintrag im Küchenbuch.*

STOLPERFALLE

Unter Argo CD gehen `helm list`, `helm rollback` und `lookup` ins Leere. Wer einen solchen Release von Hand mit Helm anfasst, kämpft gegen `selfHeal`. Im Notfall gilt: Korrigiere in Git.

IM INTERVIEW

Wie arbeitet Argo CD mit Helm?

Es rendert den Chart mit `helm template` und verwaltet die Objekte selbst. Es gibt keinen Helm-Release im Cluster. Flux dagegen führt echte Helm-Releases aus. Das ist bei Rollbacks und `lookup` wichtig.

DAS NIMMST DU MIT

- Argo CD rendert, Flux führt Helm aus.
- Rollback heißt bei GitOps: Revert in Git.
- `lookup` gehört nicht in Argo-Charts.

34

Helm oder Kustomize?

Zwei Wege zum selben Ziel, oft zusammen im Einsatz.

	Helm	Kustomize
Ansatz	Templates mit Variablen	Reines YAML mit Patches
Sprache	Go-Templates	Keine, nur YAML
Paket	Versioniert, verteilbar	Kein Paketformat
Lebenszyklus	Release, Historie, Rollback	Keiner, nur `kubectl apply -k`
Stärke	Software verteilen, viele Optionen	Eigene Manifeste je Umgebung anpassen
Schwäche	Templates werden schnell unlesbar	Ungeeignet für viele Nutzer mit vielen Optionen

> **DAS BILD IM KOPF**
>
> *Helm ist ein Formular: Der Autor bestimmt, was man ausfüllen kann. Kustomize ist ein Textmarker: Du kannst jede Stelle ändern, musst aber wissen, wo sie steht.*

Faustregel

- Du **veröffentlichst** Software für andere: Helm.
- Du passt **eigene** einfache Manifeste an: Kustomize.
- Du nutzt einen **fremden Chart** und brauchst ein Feld, das er nicht anbietet: beides.

Beides zusammen

Ein Chart bietet nicht jede Option. Kustomize kann das Ergebnis nachträglich patchen. Unter Helm 4 geht das über ein Post-Renderer-Plugin:

```
helm upgrade web ./web --post-renderer mein-kustomize-plugin
```

Eselsbrücke: Unter Helm 4 ein Plugin-Name, kein Pfad zu einem Skript.

Alternativ rendert Kustomize den Chart selbst mit dem Feld `helmCharts`. Das braucht beim Bauen `--enable-helm`.

IM INTERVIEW

Wann Helm, wann Kustomize?

Helm, wenn ich Software mit vielen Optionen paketieren, versionieren und verteilen will. Kustomize für eigene Manifeste mit kleinen Unterschieden je Umgebung. Für fremde Charts mit fehlender Option kombiniere ich beide.

DAS NIMMST DU MIT

- Helm: Paket und Lebenszyklus.
- Kustomize: Patches auf YAML.
- Post-Renderer verbinden beide.

TEIL VII

Troubleshooting

Hier trennt sich im Interview die Spreu vom Weizen.

35

Die Methode

Erst die Schicht finden, dann den Fehler.

Jeder Helm-Fehler gehört zu einer von drei Schichten:

Schicht	Merkmal	Werkzeug
Rendern	Fehler mit Dateiname und Zeile, vor jedem Kontakt zum Cluster.	`helm template --debug`
API	Der Server lehnt ab: Validierung, Rechte, unveränderliche Felder.	`--dry-run=server`
Laufzeit	Alles angenommen, aber Pods laufen nicht.	`kubectl`, Events, Logs

> **MERKE**
>
> Der erste Satz im Interview lautet: „Ich schaue zuerst, in welcher Schicht der Fehler liegt, und ändere noch nichts.“ Viele Helm-Fehler sind eigentlich Kubernetes-Fehler.

Die erste Minute

```
helm status web -n prod
```

Eselsbrücke: In welchem Zustand ist der Release?

```
helm history web -n prod
```

Eselsbrücke: Was war die letzte gute Revision? Die Spalte `DESCRIPTION` nennt oft den Grund.

```
helm get values web -n prod --all
```

Eselsbrücke: Mit `--all` alle wirksamen Werte, auch die Standards.

```
helm get manifest web -n prod
```

Eselsbrücke: Was glaubt Helm, was läuft?

```
kubectl get events -n prod --sort-by=.lastTimestamp
```

Eselsbrücke: Was sagt Kubernetes selbst?

```
helm upgrade web ./web -n prod --debug
```

Eselsbrücke: `--debug` macht die Ausgabe ausführlicher.

Häufige Meldungen

Meldung	Kapitel
`another operation ... is in progress`	36
`has no deployed releases`	36
`field is immutable`	37
`invalid ownership metadata`	38
`no matches for kind`	38
`conflict` beim Apply	38
`context deadline exceeded`	38

IM INTERVIEW

Ein Helm-Upgrade schlägt fehl. Wie gehst du vor?

Ich lese die Fehlermeldung und ordne sie ein: Rendern, API oder Laufzeit. Dann `helm status`, `helm history`, und für Laufzeitfehler Events und Logs. Ich rolle erst zurück, wenn ich den Zustand gesichert habe.

DAS NIMMST DU MIT

- Rendern, API, Laufzeit.
- `status`, `history`, `get values --all`, `get manifest`.
- Helm-Fehler sind oft Kubernetes-Fehler.

36

pending-upgrade

Der Klassiker: Helm glaubt, dass noch jemand arbeitet.

```
Error: UPGRADE FAILED: another operation
(install/upgrade/rollback) is in progress
```

Warum das passiert

Vor jeder Aktion schreibt Helm eine neue Revision mit Status `pending-upgrade`. Dieser Eintrag ist wie eine Sperre. Bei SIGINT oder SIGTERM bricht `helm upgrade` geordnet ab und markiert die Revision als `failed`. Stirbt Helm dagegen hart, bleibt der Eintrag stehen:

- Der CI-Job wurde mit SIGKILL getötet.
- Der Laptop hat die Verbindung verloren.
- Der Pod des Runners wurde hart beendet, bevor Helm fertig war.

> **DAS BILD IM KOPF**
>
> *Ein Schild „Bin gleich zurück“ an der Tür. Der Mitarbeiter ist längst nach Hause gegangen, aber das Schild hängt noch. Alle warten davor.*

Lösen

Prüfe zuerst, ob wirklich niemand mehr arbeitet.

```
helm history web -n prod
```

Eselsbrücke: Letzte Zeile `pending-upgrade`? Welche Revision war zuvor `deployed`?

```
helm rollback web 7 -n prod
```

Eselsbrücke: Der saubere Weg: Zurück zur letzten guten Revision.

Klappt das nicht, gibt es den letzten Ausweg:

```
kubectl get secrets -n prod -l owner=helm,status=pending-upgrade
```

Eselsbrücke: Finde genau das Secret mit dem falschen Status.

```
kubectl delete secret sh.helm.release.v1.web.v8 -n prod
```

Eselsbrücke: Lösche nur diese hängende Revision. Dann ist die vorherige wieder die neueste.

> **STOLPERFALLE**
>
> Prüfe vorher den Namen doppelt. Lösche nie die Secrets aller Revisionen. Ohne sie weiß Helm nichts mehr über den Release und kann ihn nicht verwalten. Im Zweifel sichere die Secrets vorher mit `kubectl get secret -o yaml`.

Vorbeugen

- Nur ein Deployment gleichzeitig pro Release, im CI als Sperre einbauen.
- `--timeout` so setzen, dass Helm selbst aufgibt, bevor das CI es abbricht.
- CI-Jobs sauber beenden lassen, nicht hart töten.

Gescheiterte Erstinstallation

`has no deployed releases` heißt: Es gibt nur gescheiterte Revisionen, keine gute. Dann gibt es nichts, wohin ein Rollback führen könnte.

```
helm uninstall web -n prod
```

Eselsbrücke: Räume auf und installiere neu. Prüfe vorher, dass keine Daten betroffen sind.

`--rollback-on-failure` bei der Installation verhindert das, weil Helm dann selbst aufräumt.

> **IM INTERVIEW**
>
> `another operation is in progress`**. Was tust du?**
>
> Ich prüfe, ob wirklich kein Prozess läuft. Dann `helm history`, und ich rolle auf die letzte `deployed`-Revision zurück. Reicht das nicht, lösche ich gezielt das Secret der hängenden Revision. Danach verhindere ich die Ursache im CI.

DAS NIMMST DU MIT

- `pending-upgrade` ist eine verwaiste Sperre.
- Erst `rollback`, dann gezielt das Secret.
- CI sperren und Timeouts sauber setzen.

37

Immutable Fields

Manche Felder darf niemand ändern, auch Helm nicht.

```
Error: UPGRADE FAILED: ... spec.selector: Invalid value:
field is immutable
```

Kubernetes schützt bestimmte Felder. Helm kann nur weitergeben, was die API sagt.

Objekt	Unveränderlich
Deployment, DaemonSet	`spec.selector`
StatefulSet	`selector`, `serviceName`, `volumeClaimTemplates`
Job	`spec.template`
Service	`clusterIP`
PVC	`storageClassName`, Verkleinern der Größe

Die häufigsten Ursachen

- Labels im Selektor haben sich geändert, etwa durch einen neuen `fullnameOverride`.
- Eine Version oder ein `helm.sh/chart`-Label steht in `matchLabels`.
- Das Chart-Major-Update stellt die Labels um.
- Ein Job wurde als normales Objekt deployt und geändert.

Lösen

Fall	Weg
Deployment	Löschen und neu erzeugen, oder neuer Objektname. Kurze Unterbrechung einplanen.
StatefulSet	`kubectl delete sts web --cascade=orphan`, dann Upgrade. Pods und PVCs bleiben.
Job	Als Hook mit `before-hook-creation`, oder Name mit Revision.

```
kubectl delete statefulset web -n prod --cascade=orphan
```

Eselsbrücke: **orphan** heißt Waise: Das Objekt geht, seine Kinder bleiben.

STOLPERFALLE

`--force-replace` löscht und erzeugt Objekte neu. Das gibt eine Unterbrechung, und bei Services kann sich die IP ändern. Es ist kein normales Upgrade-Flag.

DAS BILD IM KOPF

Der Selektor ist die Hausnummer des Deployments. Änderst du sie, findet es seine eigenen Bewohner nicht mehr. Darum verbietet Kubernetes das.

IM INTERVIEW

Ein Upgrade scheitert mit `field is immutable`. Was ist los?

Der Chart will ein geschütztes Feld ändern, meist den Selektor. Ich prüfe mit `helm diff`, was sich geändert hat. Ich mache die Änderung rückgängig, oder ich ersetze das Objekt geplant, bei StatefulSets mit `--cascade=orphan`.

DAS NIMMST DU MIT

- Selektoren sind unveränderlich.
- Keine Versionen in `matchLabels`.
- `--cascade=orphan` rettet StatefulSets.

38

Ownership und Konflikte

Wem gehört dieses Objekt, und wer darf es ändern?

Das Objekt gibt es schon

```
Error: ... exists and cannot be imported into the current
release: invalid ownership metadata
```

Helm verwaltet nur Objekte mit diesen Markierungen:

```yaml
metadata:
  labels:
    app.kubernetes.io/managed-by: Helm
  annotations:
    meta.helm.sh/release-name: web
    meta.helm.sh/release-namespace: prod
```

Ein von Hand angelegtes Objekt hat sie nicht. Du hast zwei Wege:

- Du setzt die Markierungen mit `kubectl label` und `kubectl annotate`.
- Du nutzt `--take-ownership`. Helm übernimmt das Objekt dann ohne diese Prüfung.

> **STOLPERFALLE**
>
> Zwei Releases, die dasselbe Objekt erzeugen, stören sich. Das passiert oft mit globalen Objekten wie ClusterRoles. Setze den Release-Namen in den Namen solcher Objekte.

Konflikte bei Server-Side Apply

Gehört ein Feld einem anderen Field Manager, scheitert das Apply. Finde heraus, wer das ist:

```
kubectl get deployment web -n prod -o yaml --show-managed-
fields
```

Eselsbrücke: `managedFields` zeigt, wem welches Feld gehört.

Dann entscheidest du: Soll das Feld aus dem Chart raus, oder soll Helm es mit `--force-conflicts` übernehmen?

Entfernte APIs

```
Error: ... no matches for kind "Ingress"
in version "networking.k8s.io/v1beta1"
```

Das tritt nach einem Kubernetes-Upgrade auf: Das gespeicherte Manifest nutzt eine entfernte API. Helm scheitert schon beim Lesen des alten Stands. Das Plugin `helm-mapkubeapis` schreibt das Release-Secret auf aktuelle APIs um.

> **STOLPERFALLE**
>
> `helm-mapkubeapis` baut noch auf dem Helm-3-SDK auf, und seine Doku nennt Helm 4 nicht. Ob es mit Helm 4 zuverlässig läuft, ist nicht belegt. Sichere die Release-Secrets vorher und probiere es zuerst im Testcluster.

> **MERKE**
>
> Prüfe vor jedem Kubernetes-Upgrade deine Charts mit `helm lint --kube-version` und mit Pluto: `helm template ./web | pluto detect -`. Das geht unabhängig von der Helm-Version. `pluto detect-helm` prüft laufende Releases, seine Doku nennt aber nur Helm 2 und 3.

Timeouts

`context deadline exceeded` heißt: Helm hat gewartet, aber etwas wurde nicht fertig. Helm ist selten schuld.

1. `helm status` und `kubectl get pods` zeigen, was nicht bereit ist.
2. `kubectl describe` und Logs erklären, warum.
3. Typisch: Image nicht ladbar, Probe falsch, ein Hook-Job hängt, PVC bleibt `Pending`.

> **IM INTERVIEW**
>
> **`invalid ownership metadata`: Was bedeutet das?**
>
> Das Objekt existiert bereits, gehört aber nicht zu diesem Release. Ich prüfe, wer es angelegt hat. Soll Helm es übernehmen, setze ich die Markierungen oder nutze `--take-ownership`.

IM INTERVIEW

Nach einem Kubernetes-Upgrade scheitert jedes `helm upgrade`. Warum?

Das gespeicherte Release-Manifest nutzt eine entfernte API. Der Chart muss aktualisiert werden, und das Release selbst lässt sich mit `helm-mapkubeapis` umschreiben.

DAS NIMMST DU MIT

- Helm erkennt Eigentum an Label und Annotationen.
- `managedFields` zeigt Konflikte.
- `mapkubeapis` repariert alte APIs im Release.
- Timeout heißt: Schau auf die Pods.

TEIL VIII

Das Interview

Fragen, Aufgaben und ein Spickzettel für den letzten Abend.

39

Die Fragen

Kurze Fragen, kurze Antworten. Zum Wiederholen vor dem Gespräch.

Grundlagen

Was ist Helm?

Der Paketmanager für Kubernetes: Templates plus Werte ergeben Manifeste, dazu Historie und Rollback.

Chart, Release, Repository?

Paket, installierte Instanz, Ablageort für Pakete.

Was ist eine Revision?

Ein Versionsstand eines Releases. Jedes `upgrade` und `rollback` erzeugt eine neue.

`version` oder `appVersion`?

Chart-Version und Anwendungsversion, unabhängig voneinander.

Warum gibt es Tiller nicht mehr?

Weite Rechte im Cluster. Heute nutzt Helm die RBAC-Rechte des Nutzers.

Wo läuft Helm?

Als Programm beim Client oder im CI, nicht im Cluster.

Wo speichert Helm den Zustand?

In einem Secret pro Revision, im Namespace des Releases.

Ein Chart mehrfach im selben Namespace?

Ja, mit verschiedenen Release-Namen, wenn die Objektnamen davon abhängen.

Werte

In welcher Reihenfolge gelten Werte?

`values.yaml`, Eltern-Chart, `-f` von links nach rechts, `--set`.

Wie werden Listen zusammengeführt?

Gar nicht, sie werden ersetzt. Maps werden gemergt.

Wie löschst du einen Standardwert?

Setze ihn auf `null`.

`--set` oder `--set-string`?

`--set` rät den Typ, `--set-string` erzwingt String.

Warum ist `--reuse-values` riskant?

Neue Standards des Charts fehlen.

Wozu `values.schema.json`?

Werte vor dem Rendern prüfen, inklusive Typen und Tippfehlern.

Was bedeutet `global`?

Werte, die der Eltern-Chart und alle Subcharts sehen.

Templates

Was ist der Punkt?

Der aktuelle Kontext.

Was ist `$`?

Immer die Wurzel des Templates.

`{{-` und `-}}`?

Entfernen Leerraum links oder rechts, samt Zeilenumbruch.

`indent` oder `nindent`?

`nindent` setzt zusätzlich einen Zeilenumbruch davor.

`include` oder `template`?

`include` lässt sich in eine Pipe geben.

`if` oder `with`?

Beide prüfen auf leer, `with` ändert zusätzlich den Punkt.

Was macht `required`?

Bricht mit einer Meldung ab, wenn der Wert fehlt.

Was macht `tpl`?

Rendert einen String als Template.

Was macht `lookup`?

Liest Objekte aus dem Cluster. Bei `helm template` ist es leer.

Warum `quote`?

YAML rät Typen, Strings sollen String bleiben.

Warum Präfix bei `define`?

Benannte Templates sind global, auch über Subcharts hinweg.

Warum `trunc 63`?

Viele Kubernetes-Namen dürfen nur 63 Zeichen lang sein.

Warum ändert eine ConfigMap die Pods nicht?

Das Pod-Template bleibt gleich. Eine Prüfsummen-Annotation löst das.

Warum keine Zufallsfunktion in Secrets?

Jeder Lauf ergibt einen neuen Wert, also Änderung bei jedem Upgrade.

Fortgeschrittene Charts

`dependency update` oder `build`?

`update` löst auf, `build` folgt `Chart.lock`.

Was ist ein Library Chart?

Nur Bausteine, nicht installierbar, `type: library`.

Was ist ein Hook?

Ein Objekt mit Annotation, das zu einem Zeitpunkt im Lebenszyklus läuft.

Sind Hooks Teil des Releases?

Nein. Sie brauchen eine Delete-Policy.

Wie bestimmst du die Hook-Reihenfolge?

Mit `helm.sh/hook-weight`, aufsteigend.

Was tut `crds/`?

Installiert einmal, ändert und löscht nie.

Wie aktualisierst du CRDs?

Eigener CRD-Chart oder bewusst mit `kubectl apply`.

Wie schützt du ein Objekt beim Uninstall?

Mit `"helm.sh/resource-policy": keep`.

Was ist ein Chart-Test?

Ein Pod mit `"helm.sh/hook": test`. Exit-Code 0 heißt bestanden.

Betrieb

Warum `upgrade --install`?

Idempotent, gleicher Befehl für Erstinstallation und Update.

Was macht `--rollback-on-failure`?

Rollt zurück oder räumt auf, früher `--atomic`.

Wann gilt ein Release als erfolgreich?

Ohne `--wait`, wenn die API die Objekte angenommen hat.

Was stellt ein Rollback wieder her?

Manifeste, keine Daten.

3-Way-Merge?

Vergleich von altem Manifest, Live-Stand und neuem Manifest.

Server-Side Apply?

Der API-Server führt zusammen und kennt pro Feld den Eigentümer.

Was ist neu in Helm 4?

SSA als Standard für neue Installationen, kstatus, neue Plugins, umbenannte Flags.

Warum OCI?

Eine Registry für alles, gleiche Rechte, Digests, kein Index.

Wie signierst du Charts?

`helm package --sign` mit PGP oder `cosign` in OCI.

Wie gehst du mit Secrets um?

Kein Klartext. Verweis im Chart, Daten aus External Secrets, SOPS oder Sealed Secrets.

Argo CD und Helm?

Rendert und wendet selbst an. Es gibt keinen Helm-Release.

Flux und Helm?

Führt echte Helm-Aktionen aus.

Helm oder Kustomize?

Helm für Pakete und Lebenszyklus, Kustomize für Patches an eigenem YAML.

Wenn du etwas nicht weißt

Sag es. Dann denk laut: „Das habe ich so nicht gemacht. Ich vermute X, weil Y. Ich würde mit `helm template` prüfen, was herauskommt.“ Ein Rendern ohne Risiko ist bei Helm immer eine gute Antwort.

DAS NIMMST DU MIT

- Kurze Antwort zuerst. Details nur, wenn jemand nachfragt.
- Sage dazu, was in Produktion schiefgehen kann.
- Ein eigenes Beispiel schlägt jede Definition.

40

Aufgaben

Sechs Aufgaben, die in Interviews oft kommen. Mit Lösung.

> **MERKE**
>
> Ablauf im Interview: nachfragen, laut planen, klein anfangen, rendern, verbessern. Sprich beim Tippen.

Aufgabe 1: Optionaler Ingress

Situation: Ein Ingress soll nur entstehen, wenn er aktiviert ist, mit beliebig vielen Hosts.

`values.yaml`

```yaml
ingress:
  enabled: false
  hosts: []
```

`templates/ingress.yaml`

```yaml
{{- if .Values.ingress.enabled }}
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: {{ include "web.fullname" . }}
spec:
  rules:
    {{- range .Values.ingress.hosts }}
    - host: {{ . | quote }}
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: {{ include "web.fullname" $ }}
                port:
                  number: {{ $.Values.service.port }}
    {{- end }}
{{- end }}
```

Das passiert hier:

- `if` schaltet das ganze Objekt.
- `range` setzt den Punkt auf den Host. Darum ist es dort `$`.
- `include` bekommt `$`, denn der Baustein braucht die Wurzel.

Aufgabe 2: Wenn eine ConfigMap wechselt

Situation: Pods sollen automatisch neu starten, wenn sich die Konfiguration ändert.

Lösung: Prüfsumme als Annotation im Pod-Template, siehe Kapitel 14. Sage dazu, dass das nur für Werte aus dem Chart funktioniert. Ändert jemand die ConfigMap von Hand, passiert nichts.

Aufgabe 3: Datenbank-Migration ohne Ausfall

Situation: Eine neue Version braucht ein neues Datenbankschema.

- **Hook:** `pre-upgrade`-Job mit `backoffLimit`, Delete-Policy `before-hook-creation`.
- **Abwärtskompatibel:** erst Spalten hinzufügen, nie im selben Schritt löschen. Die alte Version muss mit dem neuen Schema laufen.
- **Aufräumen:** Alte Spalten kommen erst im nächsten Release weg.
- **Rollback:** dreht nur Manifeste zurück, nie das Schema.

Aufgabe 4: 30 Microservices

Situation: Alle Teams kopieren denselben Chart.

- **Library Chart** mit Deployment, Service, Labels und Security-Standards.
- Jeder Service hat einen kleinen Chart mit `dependency` und eigenen Werten.
- Strenge SemVer-Versionen, automatische Updates per Renovate oder Dependabot.
- **Unit-Tests** für den Library Chart.
- Alternative nennen: ein einziger generischer Chart mit Werten pro Service.

Aufgabe 5: Mehrere Umgebungen

Situation: Dev, Staging und Prod.

```
web/
├── Chart.yaml
├── values.yaml          gemeinsame Standards
└── values/
    ├── dev.yaml
    ├── staging.yaml
    └── prod.yaml
```

- Derselbe Chart, gleiche Version, unterschiedliche Werte.
- Die gleiche Version wandert durch alle Stufen. Du baust nicht pro Umgebung neu.
- Geheimnisse liegen nie in diesen Dateien.
- Im CI jede Umgebung rendern und prüfen.

Aufgabe 6: Das Upgrade hängt in Produktion

Situation: Die Pipeline wurde abgebrochen, ein neues Upgrade meldet `another operation is in progress`.

1. Sicherstellen, dass kein anderer Lauf aktiv ist.
2. `helm history`: Die letzte gute Revision finden.
3. `kubectl get pods`: Läuft die Anwendung überhaupt noch? Das hat Vorrang.
4. `helm rollback` auf die letzte gute Revision.
5. Nur wenn das scheitert, das Secret der `pending`-Revision löschen, nach einer Sicherung.
6. Ursache beheben: CI-Sperre, Timeouts, sauberes Beenden.

IM INTERVIEW

Wie würdest du einen Chart für ein fremdes Team gestalten?

`values.yaml` mit Kommentaren und sicheren Standards, Schema, verständliche Fehlermeldungen mit `required`, `existingSecret`-Muster, Beispiele in `README` und `NOTES.txt` und Unit-Tests für jede Option.

DAS NIMMST DU MIT

- Klein anfangen und rendern.
- `$` in `range` nicht vergessen.
- Rollback ist nicht das Ende der Geschichte.

41

Spickzettel

Alles Wichtige auf ein paar Seiten, für den letzten Abend.

Der eine Satz zu allem

Begriff	Merksatz
Chart	Das Rezept.
Release	Das gekochte Gericht.
`values.yaml`	Rechts gewinnt.
Punkt	Wo du gerade bist.
`$`	Die Zuhause-Taste.
`{{-`	Der Staubsauger.
Hook	Crew außerhalb des Programmhefts.
`crds/`	Einmal rein, nie wieder angefasst.
`pending-upgrade`	Das verwaiste Türschild.
Rollback	Manifeste, keine Daten.

Lebenszyklus

```
helm create web
helm template web ./web
helm lint ./web --strict
helm upgrade --install web ./web -n demo --create-namespace
helm list -A
helm status web -n demo
helm history web -n demo
helm rollback web 1 -n demo
helm uninstall web -n demo
```

Inspektion

```
helm get values web -n demo --all
helm get manifest web -n demo
helm get hooks web -n demo
helm show values oci://registry.example.com/charts/web
```

```
helm template web ./web -s templates/service.yaml
helm template web ./web --debug
```

Pakete

```
helm dependency update ./web
helm dependency build ./web
helm package ./web
helm registry login registry.example.com
helm push web-1.4.2.tgz oci://registry.example.com/charts
helm pull oci://registry.example.com/charts/web --untar
```

Template-Muster

```
{{- with .Values.nodeSelector }}
nodeSelector:
  {{- toYaml . | nindent 2 }}
{{- end }}

labels:
  {{- include "web.labels" . | nindent 4 }}

image: "{{ .Values.image.repository }}:{{ .Values.image.tag
  | default .Chart.AppVersion }}"

host: {{ required "host fehlt" .Values.host | quote }}

{{- range $k, $v := .Values.env }}
- name: {{ $k }}
  value: {{ $v | quote }}
{{- end }}
```

Flags, die du kennen musst

Flag	Bedeutung
`--wait`	Warten mit kstatus (`watcher`).
`--timeout 5m`	Grenze fürs Warten. Standard: 5 Minuten.
`--rollback-on-failure`	Früher `--atomic`.
`--force-replace`	Früher `--force`. Löscht und erzeugt neu.
`--force-conflicts`	Übernimmt Felder bei SSA.
`--server-side`	`true`, `false`, `auto`.
`--take-ownership`	Übernimmt bestehende Objekte.
`--dry-run=server`	Simuliert gegen den Cluster.
`--history-max`	Revisionen behalten. Standard: 10.
`--skip-crds`	Keine CRDs installieren.
`--no-hooks`	Keine Hooks ausführen.

Niemals ohne Nachdenken

- Passwörter in `values.yaml` im Git.
- Versionen in `matchLabels`.
- `--reuse-values` in einer Pipeline.
- Release-Secrets löschen, ohne Sicherung.
- `--force-replace` als Standard-Flag.
- Chart ohne fest gepinnte Version installieren.
- Zufallswerte ohne `lookup` in Secrets.

DAS NIMMST DU MIT

- Rendere zuerst.
- Werte im Git, Geheimnisse außerhalb.
- Der Cluster zählt, nicht das Template.

Stichwörter

$ 36
3-Way-Merge 67
_helpers.tpl 38
appVersion 22
Argo CD 86
.Capabilities 27
Chart 12
.Chart 27
chart-testing 84
Chart.lock 50
Chart.yaml 20
checksum/config 41
cosign 74
CRD 59
crds/ 59
default 40
define 38
Drift 68
existingSecret 76
External Secrets 76
field is immutable 97
.Files 44
Flux 86
--force-conflicts 67
global 50
Helm 10
Helm 4 70
helm install 16
helm lint 80
helm test 57
helm upgrade 62
helm-diff 64
helm-unittest 82
helm.sh/hook 55
helm.sh/release.v1 65
helm.sh/resource-policy 60
HELM_DRIVER 65
.helmignore 20
HelmRelease 86
Hook 55
include 38
kind 14
kstatus 70
kubeconform 80
Kustomize 89
Library Chart 53
lookup 42
mapkubeapis 99
nindent 32
OCI 72
pending-upgrade 94
Pipeline 30
Post-Renderer 89
Provenance 74
quote 30
range 34
Release 12
.Release 27
Repository 12
required 42
Revision 12
--rollback-on-failure 62
Sealed Secrets 76
SemVer 22
Server-Side Apply 67
--set 24
SOPS 76
Sprig 40
Subchart 50
--take-ownership 99
Tiller 11
toYaml 32
tpl 42
--values 24
.Values 27
values.schema.json 48
values.yaml 20
--wait 62
with 34