Parametrisierte Kubernetes Deployments mit kustomize

kustomize ermöglicht die Anpassung von Kubernetes-Deployments zur Erzeugung mehrerer Varianten (z. B. für unterschiedliche Umgebungen). Dabei werden die originalen YAML-Dateien nicht modifiziert, sondern mit Overlays überlagert. Im Gegensatz zu Helm kommt kustomize also ganz ohne Templates aus, was die Verwendung besonders einfach macht. In diesem Blog-Post stellen wir die wichtigsten Features von kustomize anhand eines Beispiels vor.

Jegliche Ressourcen in Kubernetes werden als YAML-Dateien definiert und per „kubectl apply“ an den API-Server geschickt. Nun dauert es nicht lange, bis von einem Deployment leicht unterschiedliche Varianten notwendig werden. Im einfachsten Fall wird in PROD eine andere Datenbank-URL benötigt als in DEV. Komplexer wird es, wenn die YAML-Dateien an andere User ausgeliefert werden (z.B. in OpenSource-Projekten) und so eigentlich jede Einstellung anpassbar sein soll.

Diese „Tooling-Lücke“ hat lange Zeit „Helm“ gefüllt und sich zum De-facto-Standard entwickelt, der nun zu wackeln beginnt. Warum? Gerade in Deployment-Szenarien mit überschaubarer Komplexität macht Helm die Wartbarkeit unnötig schwer, bremst die Entwicklung und erhöht die Fehleranfälligkeit. In diesem Blog-Post schauen wir uns daher kustomize als Alternative an, welche kürzlich auch in die Kubernetes-CLI kubectl integriert wurde. Um das Fazit vorwegzunehmen: Ich würde kustomize insbesondere für kleinere Projekte empfehlen!

DAS ORIGINAL

Schauen wir uns die Alternativen Schritt für Schritt an. Ein simples Deployment eines Alpine-Linux sieht bspw. so aus:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app
        image: alpine:3.10
        tty: true
        stdin: true
```

```
        env:
          - name: foo
            value: bar
        resources:
          limits:
            memory: "64Mi"
            cpu: "100m"
```

Hier ist nichts parametrisiert - diese YAML-Datei ist konform zur Kubernetes API-Spezifikation und wird somit direkt vom API-Server akzeptiert. Schreibt man eine solche YAML-Datei in einer IDE wie Visual Studio Code oder IntelliJ, kann man sich über Features wie Auto-Completion freuen.

HELM

Helm nutzt das Templating-System von Go und bündelt mehrere zusammengehörige Template-Dateien zu sog. Charts - häufig bleibt es ja nicht bei einem Deployment, sondern es gesellen sich noch weitere YAML-Dateien für Services, Ingresses, Secrets usw. hinzu, bevor daraus ein fachlich vollständiges Deployment-Paket wird. Mit dem Kommando helm create <app-name> kann man sich eine Vorlage erzeugen und diese weiter ausbauen. Wie sieht das Pendant für unser Alpine-Deployment aus?

```
# File: helm/my-app/templates/deployment.yaml

apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "my-app.fullname" . }}
  labels:
{{ include "my-app.labels" . | indent 4 }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app.kubernetes.io/name: {{ include "my-app.name" . }}
      app.kubernetes.io/instance: {{ .Release.Name }}
  template:
    metadata:
      labels:
        app.kubernetes.io/name: {{ include "my-app.name" . }}
        app.kubernetes.io/instance: {{ .Release.Name }}
    spec:
    {{- with .Values.imagePullSecrets }}
      imagePullSecrets:
        {{- toYaml . | nindent 8 }}
    {{- end }}
      containers:
        - name: {{ .Chart.Name }}
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
```

```
          imagePullPolicy: {{ .Values.image.pullPolicy }}
          tty: {{ .Values.tty }}
          stdin: {{ .Values.stdin }}
          env:
          {{- toYaml .Values.env | nindent 12 }}
          resources:
            {{- toYaml .Values.resources | nindent 12 }}
      {{- with .Values.nodeSelector }}
      nodeSelector:
        {{- toYaml . | nindent 8 }}
      {{- end }}
    {{- with .Values.affinity }}
      affinity:
        {{- toYaml . | nindent 8 }}
    {{- end }}
    {{- with .Values.tolerations }}
      tolerations:
        {{- toYaml . | nindent 8 }}
    {{- end }}
```

Lesbarkeit? Hm.

IDE-Unterstützung wie Auto-Completion? Nicht mehr vorhanden, bestenfalls rudimentär.

Ach so, zu der Template-Datei gehört natürlich noch eine values.yaml-Datei mit den Werten für die Platzhalter:

```
# File: helm/my-app/values.yaml

replicaCount: 1

image:
  repository: alpine:3.10
  tag: stable
  pullPolicy: IfNotPresent

tty: true
stdin: true

env:
  - name: foo
    value: bar

resources:
  limits:
    cpu: 100m
    memory: 64Mi
```

Eigentlich haben wir jetzt - neben dem sperrigen Template - eine values.yaml-Datei, die fast genauso groß ist wie das Original. Hm.

Das Problem der Wartbarkeit kann man sich an dem kleinen Beispiel bereits vorstellen, wächst aber mit der Größe des jeweiligen Projektes. Das Deployment unserer vPW-Showcase-Umgebung besteht bspw. aus knapp 90 Template-Dateien mit über 3000 Zeilen Code, wobei externe Abhängigkeiten wie Elasticsearch, Kafka und Co hier noch nicht mit eingerechnet sind. Was in überschaubaren Deployments „nur“ den Spaß der Entwickler trübt, wird bei größeren Setups zu einem relevanten Risiko, denn die Fehleranfälligkeit nimmt extrem zu.

KUSTOMIZE

Was macht kustomize nun anders?

- **Keine Templates:** kustomize setzt auf eine Overlay-/Patch-Strategie, d. h. man schreibt „normale“ YAML-Dateien, die durch Overlays ergänzt werden (also keine Templates mit Platzhaltern wie bei Helm!)
- **Deklarativ:** Da man jederzeit valide Kubernetes-YAML-Dateien ohne Platzhalter schreibt, hat man automatisch die volle IDE-Unterstützung inkl. Auto-Completion.
- **Praktische Features:** Typische Use-Cases wie „alle Ressourcen sollen ein bestimmtes Label erhalten und in einen angegebenen Namespace deployed werden“ sowie „Änderungen einer Config-Map sollen zu einem neuen Rollout des Deployments führen“ werden gezielt adressiert.
- **Flexibel und unabhängig:** Das CLI-Tool kustomize rendert letztlich nur YAML-Output nach Stdout, welcher anschließend je nach Geschmack mittels des Bash-Pipe-Operators weiterverarbeitet werden kann (z. B. kustomize build ~/someApp/overlays/production | kubectl apply -f -)
- **Integration:** Seit kubectl v.1.14 sind die kustomize-Funktionen für das Rendering in kubectl integriert, sodass keine weitere Binary benötigt wird.

OVERLAYS

Statt Templates verwendet kustomize sog. Overlays. Die Idee: Man definiert eine Basis-Version der YAML-Dateien („base“) und dann Overlays für Modifikationen des Originals. Dabei handelt es sich um normale YAML-Dateien, welche nur die Attribute der „spec“ enthalten, welche dann das Original überdecken/überschreiben sollen. In folgender Abbildung wird bspw. der Replica-Count eines Deployments von 1 auf 3 geändert. Neben den abweichenden Attributen muss das Overlay auch die notwendigen Informationen enthalten, um die Ressource zu identifizieren, die überlagert werden soll (hier also ein Deployment mit Namen „my-app“). Weitere Details zu diesem Mechanismus schauen wir uns im Folgenden an.

From:
https://www.cooltux.net/ - **TuxNet DokuWiki**

Permanent link:
https://www.cooltux.net/doku.php?id=it-wiki:kubernetes:deployments_mit_kustomize&rev=1710329278

Last update: **2024/03/13 11:27**