

corewire

Helm

Helm Charts erstellen

# Folien-Hinweis

- Space, Page down: Nächste Folie
- Page up: Vorherige Folie
- ESC, o: Übersicht

[Zur Kapitelübersicht](#)

# Lernziele

- Ein neues Chart mit sauberem Grundgerüst erstellen
- Chart-Metadaten, Values und Templates sinnvoll strukturieren
- Maps in `values.yaml` modellieren und in Templates rendern
- Maps mit `range` iterieren, z. B. für Umgebungsvariablen
- Typische Template-Funktionen für Wiederverwendung und Bedingungen nutzen
- Ein Chart lokal prüfen, rendern und paketieren

# Ausgangspunkt: Neues Chart

```
helm create webshop
```

- Es wird ein vollständiges Chart-Grundgerüst erstellt
- Beispiel-Templates sind direkt enthalten
- Das Gerüst kann als Startpunkt für eigene Apps verwendet werden

# Chart-Struktur verstehen

```
webshop/  
  Chart.yaml      # Metadaten zum Chart  
  values.yaml     # Standardwerte für Templates  
  charts/         # Abhängigkeiten (Subcharts)  
  templates/      # Kubernetes-Manifest-Templates  
    _helpers.tpl  # Hilfsfunktionen  
    deployment.yaml  
    service.yaml  
    ingress.yaml
```

# Chart-Metadaten in `Chart.yaml`

```
apiVersion: v2
name: webshop
description: Helm Chart für die Webshop-Anwendung
type: application
version: 0.1.0
appVersion: "1.0.0"
```

- `version`: Version des Charts
- `appVersion`: Version der ausgerollten Anwendung

# Defaults in `values.yaml`

```
replicaCount: 2

image:
  repository: ghcr.io/example/webshop
  tag: "1.0.0"

service:
  type: ClusterIP
  port: 80
```

`values.yaml` ist die zentrale Stelle für konfigurierbare Standardwerte.

# Templates mit Values

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "webshop.fullname" . }}
spec:
  replicas: {{ .Values.replicaCount }}
  template:
    spec:
      containers:
        - name: webshop
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
```



# Maps in `values.yaml`

```
labels:  
  app.kubernetes.io/name: webshop  
  app.kubernetes.io/component: frontend  
  
resources:  
  requests:  
    cpu: "100m"  
    memory: "128Mi"  
  limits:  
    cpu: "500m"  
    memory: "512Mi"
```

Maps sind Key-Value-Strukturen und eignen sich für konfigurierbare Felder mit mehreren Unterwerten.

# Maps im Template rendern

```
metadata:  
  labels:  
  {{- toYaml .Values.labels | nindent 4 }}  
  
spec:  
  containers:  
    - name: webshop  
      resources:  
  {{- toYaml .Values.resources | nindent 8 }}
```

Mit `toYaml` und `nindent` werden Maps sauber und korrekt eingerückt ausgegeben.

# Map mit `range` iterieren (Env-Variablen)

Values als Map:

```
env:  
  LOG_LEVEL: info  
  FEATURE_FLAG_X: "true"  
  HTTP_PORT: "8080"
```

Template im Container:

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

So lassen sich beliebig viele Umgebungsvariablen dynamisch aus einer Map erzeugen.

# Wiederverwendung mit `_helpers.tpl`

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

## Vorteile:

- Namenskonventionen zentral gepflegt
- Weniger Copy/Paste in mehreren Templates
- Einheitliche Ressourcen-Namen

# Bedingte Ressourcen

```
{{- if .Values.ingress.enabled }}  
apiVersion: networking.k8s.io/v1  
kind: Ingress  
metadata:  
  name: {{ include "webshop.fullname" . }}  
spec:  
  rules:  
    - host: {{ .Values.ingress.host | quote }}  
{{- end }}
```

So bleiben Charts flexibel für unterschiedliche Umgebungen.

# Qualität sicherstellen

## Template-Syntax und Best Practices prüfen

```
helm lint ./webshop
```

## Templates lokal rendern

```
helm template dev-webshop ./webshop
```

## Installationslauf simulieren

```
helm install dev-webshop ./webshop --dry-run --debug
```

# Chart paketieren

Abhängigkeiten aktualisieren

```
helm dependency update ./webshop
```

Chart als Archiv bauen

```
helm package ./webshop
```

Ergebnis: `webshop-0.1.0.tgz`

# Empfohlener Erstellungs-Workflow

1. `helm create` als Startpunkt
2. Beispiel-Templates auf die eigene App reduzieren
3. Konfiguration in `values.yaml` sauber modellieren
4. Namen und Labels über `_helpers.tpl` zentralisieren
5. Mit `helm lint` und `helm template` früh prüfen
6. Erst danach paketieren und verteilen



# Demo: Eigenes Helm Chart bauen

# Lab

- Gehen Sie auf: <https://labs.corewire.de>
- Navigieren Sie nach:
  - Helm
  - Aufgaben
  - Helm Charts

## Zur Kapitelübersicht

- Vorheriges Kapitel: [Helm Values](#)
- Nächstes Kapitel: [CI/CD und GitOps](#)