Service Deployment im KoP (OCP4)

Einleitung

Glossar

Vom Code zu Container

Bevor wir uns mit dem eigentlichen Deployment beschäftigen, ist es wichtig zu verstehen, wie aus unserem Code ein lauffähiger Container wird. Man kann sich diesen Kontext wie eine **Werkstatt** vorstellen: Vor der Auslieferung zum Kunden (Deployment) müssen die Produkte erst produziert (**Build-Prozess**) und verpackt werden (**Containerisierung**), damit sie einfach zu liefern sind.

Build-Prozess:

Nachdem wir unseren Code geschrieben haben, nutzen wir Build-Tools wie Gradle für Java-Projekte oder npm für JavaScript-Anwendungen. Diese Tools kompilieren unseren Code und erstellen ein sogenanntes Artefakt – das ist im Grunde das fertige Programm, das ausgeführt werden kann.

Verpackungs-Prozess:

Dieses Artefakt wird dann in ein **Docker-Image** verpackt. Die Verpackung erfolgt in ein vorbereitetes Image, das eine minimalistische Laufzeitumgebung für unsere Anwendung enthält. In der Regel ist das ein einfacher Linux-Kernel und ein JVM für eine Java-Anwendung. Nach der Verpackung enthält das Image alles, was die Anwendung zum Laufen braucht: den Code selbst, alle Bibliotheken, Konfigurationen und die Teile des Betriebssystems.

Dieses Image wird später auf einer Container-Plattform (wie Docker Desktop, Kubernetes oder OpenShift) gestartet und läuft dort als separater Container.

Kubernetes als Container-Platform

Im OCP4 nutzen wir **Kubernetes** als **Container Plattform** für unsere Container. Kubernetes übernimmt dabei viele wichtige Aufgaben:

- **Deployment**: Kubernetes startet unsere Container und sorgt dafür, dass sie laufen
- **Scaling**: Bei hoher Last werden automatisch mehr Container gestartet
- **Self-Healing**: Stürzt ein Container ab, wird er automatisch neu gestartet
- **Load Balancing**: Anfragen werden gleichmäßig auf alle Container verteilt
- **Service Discovery**: Services können sich gegenseitig finden, ohne feste IP-Adressen zu kennen

Deployment

Das primäre Kubernetes-Objekt hierfür ist das **Deployment**. Ein Deployment beschreibt den gewünschten Zustand für eine Anwendung. Es sorgt dafür, dass eine bestimmte Anzahl von Pod-Replikaten läuft und kümmert sich um Updates (z. B. bei einer neuen Image-Version) durch Strategien wie Rolling Updates.

- **Zuständiges Objekt:** Deployment

Scaling

Für die Skalierung sind zwei Objekte zentral:

- **Deployment/ReplicaSet**: Hier wird die *Anzahl* der gewünschten Pods (Replicas) festgelegt. Eine manuelle Skalierung erfolgt durch die Änderung dieser Anzahl.
- **Zuständige Objekte:** Deployment

Self-Healing

Die Selbstheilung ist eine Kernfunktion, die durch die Kombination von Controllern sichergestellt wird. Fällt ein Pod aus, bemerkt der Controller des **Replica Set** (das vom **Deployment** verwaltet wird), dass die tatsächliche Anzahl der Pods nicht mehr dem gewünschten Zustand entspricht. Daraufhin startet er automatisch einen neuen Pod.

- **Zuständige Objekte:** Deployment / ReplicaSet

Load Balancing & Service Discovery

Beide Aufgaben werden hauptsächlich vom **Service**-Objekt übernommen.

- **Load Balancing**: Ein Service verteilt den Netzwerkverkehr (die Anfragen) auf alle Pods, die zu diesem Service gehören. So wird die Last gleichmäßig verteilt.
- **Service Discovery**: Ein Service erhält eine stabile, interne IP-Adresse und einen DNS-Namen. Andere Anwendungen im Cluster können diesen DNS-Namen verwenden, um den Service zu finden, unabhängig davon, auf welchen Nodes die einzelnen Pods gerade laufen oder welche IP-Adressen sie haben.

Für Anfragen von außerhalb des Clusters kommt in OpenShift zusätzlich oft eine **Route** (oder in Standard-Kubernetes ein **Ingress**) zum Einsatz, die den externen Traffic an den internen Service weiterleitet.

Zuständige Objekte: Service (und Route / Ingress für externen Zugriff)

Kubernetes-Objekte

Damit unsere Anwendung in Kubernetes laufen kann, brauchen wir mindestens vier wichtige Bausteine:

1. Namespace - Der Arbeitsbereich

Ein Namespace ist wie ein eigener Bereich innerhalb von **Kubernetes**. Er trennt verschiedene **Bounded-Kontext** voneinander, damit sie sich nicht gegenseitig stören. In unserem Fall haben wir zwei Hauptbereiche: **elpa-elpa4** für die ELPA4-Services und **elpa-elpaclassic** für die alte System. Alle Ressourcen einer Anwendung werden in ihrem Namespace gruppiert.

2. Deployment - Die Anwendungsdefinition

Das Deployment ist das Herzstück unserer Anwendung. Es beschreibt, wie unsere Container laufen sollen. Schauen wir uns die wichtigsten Teile anhand unseres envs/base/deployment.yaml an:

Container-Definition: Hier wird festgelegt, welches Docker Image verwendet wird und welche Ports geöffnet werden.

envs/base/deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app
  annotations:  # Zusätzliche Informationen für andere Tools
    sidecar.istio.io/inject: "true"  # Aktiviert Service Mesh
spec:
  replicas: 1  # Anzahl der Container-Instanzen
  template:
    spec:
      containers:
        - name: app
          image: app-image  # Das Docker Image
          ports:
            - containerPort: 8080  # Port für die API
```

Health Checks: Kubernetes prüft regelmäßig, ob unsere Anwendung noch funktioniert:

envs/base/deployment

```
readinessProbe: # Ist die App noch am Leben?
  httpGet:
    path: /health/readiness
    port: 8081
  periodSeconds: 20
  timeoutSeconds: 5
livenessProbe:
  httpGet:
    path: /health/liveness
    port: 8081
  periodSeconds: 15
  failureThreshold: 5
startupProbe: # Ist die App bereit für Anfragen?
  httpGet:
    path: /health/readiness
    port: 8081
  failureThreshold: 9
  periodSeconds: 10
  timeoutSeconds: 5
```

Umgebungsvariablen: Die Anwendung bekommt ihre Konfiguration über zwei Wege:

envs/base/deployment

```
envFrom:
  - configMapRef:
      name: backend-cm  # Lädt alle Variablen aus der ConfigMap
env:
  - name: K8S_POD_NAME  # Einzelne Variable
    valueFrom:
      fieldRef:
        fieldPath: metadata.name  # Holt den Pod-Namen
```

3. ConfigMap - Die Konfigurationszentrale

ConfigMap (envs/envs/base/configmap.yaml) speichert alle Einstellungen-Wert bzw Umgebung-Variable , die unsere Anwendung braucht:

envs/base/configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: backend-cm
data:
  LOG_LEVEL: "INFO"  # Wie viel soll geloggt werxden?
  ENABLE_JSON_LOGGING: "true"  # Format der Logs
  POSTGRES_SSL_MODE: "prefer"  # Datenbankverbindung
```

Diese Werte werden als **Umgebungsvariablen in den Container** geladen. Der Vorteil: Wir können die Konfiguration ändern, ohne ein neues Docker Image bauen zu müssen. Für die Spring-Boot-Anwendung ist eine ConfigMap nützlich, um die Werte der Variablen in der `application.yaml` zu überschreiben.

4. Service

Ein **Service** ist ein Kubernetes-Objekt, das eine **stabile Netzwerkschnittstelle** (IP + DNS-Name) für eine Gruppe von Pods bereitstellt.

Da Pods kurzlebig sind (neue IP bei Restart/Scaling), sorgt der Service für eine **dauerhafte Erreichbarkeit**.

Im Kubernetes-Projekt verwenden wir derzeit ausschließlich den Standard-Service-Typ **`ClusterIP`**.

Dieser reserviert eine konstante IP-Adresse, die nur innerhalb des Clusters erreichbar ist. Sobald der Service in Kubernetes angelegt wird, wird automatisch ein DNS-Eintrag erstellt (**FQDN**: `<service-name>.<namespace>.svc.cluster.local`).

envs/base/service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: service
spec:
  selector:
    app: hint
  ports:
    - name: http        # Die Definition eines Port-Namens ist sinnvoll, besonders bei mehreren Ports pro App,
da er eine bessere Referenz als die Portnummer darstellt (z.B. im Ingress).
      port: 8080               # Port im Service
      targetPort: 8080  # Port im Pod-Container
      appProtocol: http
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app
  labels:
      app: hint
spec:
  selector:
    matchLabels:
      app: hint
  template:
    metadata:
      labels:
        app: hint
```

In der Service-Definition müssen keine Informationen zu IP oder Domain-Namen hinterlegt werden, da Kubernetes diese automatisch generiert und verwaltet. Diese Werte bleiben bestehen, solange der Service im Cluster existiert.

Das Feld `**selector**` ist für Services von großer Bedeutung. Die Schlüssel-Wert-Paare des Selektors müssen mit den **Label-Paaren** der **Ziel-Pods**(was durch Deployment definiert und verwaltet werden) übereinstimmen. Der Schlüssel muss nicht zwingend "**app**" sein; es können beliebige Paare verwendet werden (z.B. `elpaService: hint`). Diese Paare dienen Kubernetes als "Flag", um zu identifizieren, welche Pods mit dem Service verknüpft werden sollen.

Die goldene Regel lautet: Die Labels im spec.selector.matchLabels des Deployments **müssen exakt** mit den Labels im spec.template.metadata.labels (der Pod-Vorlage) übereinstimmen.

Kubernetes & Kustomize

1. Was ist Kustomize

Kustomize ist ein Kubernetes-natives Tool zur Konfigurationsverwaltung, das es ermöglicht, Kubernetes-Ressourcen deklarativ anzupassen, ohne die ursprünglichen YAML-Dateien direkt zu verändern. Es arbeitet nach dem Prinzip der Überlagerung (Overlay), bei dem eine Basis-Konfiguration durch verschiedene Anpassungsschichten erweitert oder modifiziert werden kann.

Kernkonzepte von Kustomize:

- **Bases**: Grundlegende, wiederverwendbare Kubernetes-Ressourcen
- **Overlays**: Umgebungsspezifische Anpassungen (dev, abn, prod)
- **Components**: Wiederverwendbare Konfigurationsbausteine
- **Patches**: Gezielte Modifikationen bestehender Ressourcen
- **Transformers**: Automatische Anpassungen wie Präfixe, Suffixe oder Labels

2. Projekt Struktur mit Kustomize

Das Projekt hat zwei Hauptverzeichnisse: **envs** und **kafka**.

- **kafka-Verzeichnis:** Dieses Verzeichnis dient ausschließlich der Definition von Kafka-Topics. Es wird separat verwaltet, da die Topics in einem anderen Namespace laufen und teilweise von externen Teams betreut werden. Neue Topics können hier einfach und isoliert hinzugefügt werden.
- **envs-Verzeichnis:** Dies ist das zentrale Verzeichnis für das Deployment der ELPA4-Anwendungen. Es beinhaltet die Konfigurationen für alle Umgebungen und wird mittels Kustomize bearbeitet, um die finalen Deployment-Output-Dateien zu erzeugen

In diesem Projekt wird Kustomize für zwei Hauptzwecke eingesetzt:

1 Portable Komponenten-Konfiguration

Im envs-Ordner sind wiederverwendbare Komponenten für verschiedene Infrastruktur-Services definiert:

- **Kafka**: Message-Broker-Konfiguration
- **PostgreSQL**: Relationale Datenbank-Konfiguration
- **MongoDB**: NoSQL-Datenbank-Konfiguration

Diese Komponenten fungieren als modulare Bausteine. Wenn ein Service eine dieser Komponenten benötigt, muss lediglich im kustomization.yaml des jeweiligen Services das components-Feld entsprechend deklariert werden. Dadurch erhält der Service automatisch:

- Alle erforderlichen Umgebungsvariablen
- Passende Labels und Selektoren
- Vorkonfigurierte Verbindungsparameter

2 Staging und Templating

Durch die Verwendung von Präfixen und Suffixen ermöglicht Kustomize die Verwaltung mehrerer Service-Instanzen in verschiedenen Umgebungen. Da Kubernetes-Objektnamen innerhalb eines Namespaces eindeutig sein müssen, werden durch namePrefix und nameSuffix automatisch umgebungsspezifische Namen generiert. Dies erlaubt es, die gleiche Struktur und Templates für verschiedene Stages (dev, staging, production) zu verwenden.

Die envs Verzeichnisstruktur

Die Konfiguration innerhalb des envs-Verzeichnisses folgt einem Overlay-Prinzip, das von Kustomize gesteuert wird. Es ist in eine base-Konfiguration und umgebungsspezifische Verzeichnisse wie dev unterteilt. Zukünftige Umgebungen (abn, prod) werden nach demselben Muster aufgebaut.

elpa-elpa4/envs

```
envs
 base                                                    # Basis-Konfiguration
    components                                        # Wiederverwendbare Komponenten (base/components)
       kafka
       mongo
       postgres
    configmap.yaml
    deployment.yaml
    kustomization.yaml
    service.yaml.
    virtualservice.yaml.
 dev                                                       # Overlay-Konfiguration (dev)
    components                                        # Umgebungsspezifische Komponenten (dev/components)
       kafka
       mongo
       postgres
    deployment-patch.yaml
    hint-service                                    # Anpassungen für grundlegende Baustein
       deployment-patch.yaml
       kustomization.yaml                     # Service-spezifische Konfiguration (hint-service)
       service-app-patch.yaml
       virtualservice-patch.yaml
    kustomization.yaml
 kustomization.yaml
---
kafka
 kustomization.yaml
 values.yaml
```

Die Basis-Konfiguration

Das **base**-Verzeichnis enthält die grundlegenden und abstrakten Kubernetes-Manifeste, die für alle Umgebungen gelten. Diese Dateien dienen als Vorlage (Template).

- **Inhalt:** Es deklariert Standard-Kubernetes-Objekte wie Deployment, ConfigMap, Service und VirtualService.
- **Abstraktion:** Die Konfigurationen sind generisch gehalten und auf eine typische Spring Boot-Anwendung zugeschnitten (z. B. Port 8080 für die API und 8081 für Health-Checks).
- **Platzhalter:** Felder wie metadata.name oder spec.template.spec.containers[0].image sind bewusst als Platzhalter angelegt. Sie werden in den umgebungsspezifischen Overlays mit konkreten Werten überschrieben oder ergänzt.

base/components

Dieses Unterverzeichnis definiert wiederverwendbare Konfigurations-Patches für angebundene Dienste wie **Postgres**, **MongoDB** und **Kafka**. Jede Komponente enthält ein deployment-patch.yaml, das zwei Hauptaufgaben erfüllt:

- **Labeling:** Es fügt den Dienst spezifische Labels hinzu. Diese matchLabels ermöglichen es, später gezielte Operationen auf die entsprechenden Pods anzuwenden.
- **Environment-Variablen:** Es ergänzt die notwendigen Umgebungsvariablen, die von der Anwendung für die Verbindung mit dem jeweiligen Dienst benötigt werden.

Die Overlay-Konfiguration

Verzeichnisse wie dev enthalten die spezifischen Anpassungen für die jeweilige Zielumgebung. Kustomize verwendet diese Overlays, um die base-Konfiguration zu modifizieren und eine finale, umgebungsspezifische Manifest-Sammlung zu erstellen.

Die kustomization.yaml auf der obersten Ebene des dev-Verzeichnisses wendet globale Regeln für alle Objekte dieser Umgebung an.

- **namePrefix**: Fügt allen erstellten Kubernetes-Objekten ein Präfix hinzu (z. B. dev-). Dies ist entscheidend, um Namenskonflikte zu vermeiden, falls mehrere Umgebungen im selben Cluster oder Namespace bereitgestellt werden.
- **Labeling**: Durch **Patches** wird ein einheitliches Label wie environment: dev gesetzt. Dies erleichtert die Identifizierung und Selektion aller Ressourcen, die zu einer bestimmten Umgebung gehören.

Beispiel: envs/dev/kustomization.yaml

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

# Fügt allen Ressourcen das Präfix "dev-" hinzu
namePrefix: "dev-"
# Bindet die Konfigurationen der einzelnen Services ein
resources:
  - hint-service
  # - anderer-service

# Wendet einen globalen Patch an, um das Umgebungslabel zu setzen
patches:
  - path: deployment-patch.yaml
```

Beispiel für den zugehörigen Patch (deployment-patch.yaml):

```
apiVersion: apps/v1
kind: Deployment

metadata:
  name: app # Name wird von Kustomize mit dem aus der Base gematcht
  labels:
    environment: dev

spec:
  selector:
    matchLabels:
      environment: dev
  template:
    metadata:
      labels:
        environment: dev
```

Umgebungsspezifische Komponenten (dev/components)

Analog zum base-Verzeichnis existiert auch im dev-Overlay ein components-Verzeichnis. Dessen Zweck ist es, umgebungsspezifische Konfigurationen – primär Umgebungsvariablen – bereitzustellen. Beispielsweise werden hier die Verbindungsdaten zur dev-Postgres-Datenbank definiert, die sich von denen der Abnahme- oder Produktionsumgebung unterscheiden.

Um diese Variablen zu überschreiben oder hinzuzufügen, wird die merge-Operation des configMapGenerator genutzt. Passwörter und andere sensible Daten werden über Patches aus Kubernetes Secrets geladen.

Beispiel: envs/dev/components/postgres/kustomization.yaml

```
apiVersion: kustomize.config.k8s.io/v1alpha1
kind: Component

# Fügt Werte zur ConfigMap hinzu oder überschreibt bestehende
configMapGenerator:
- name: backend-cm
  behavior: merge
  literals:
    - POSTGRES_URL=jdbc:postgresql://vipsiae11t.system-a.local:5432/elpapgt
    - POSTGRES_USER=ZA-SVC-ELPAPG-T

# Patch, um das Passwort aus einem Secret zu laden
patches:
  - path: deployment-patch.yaml
    target:
      kind: Deployment
      name: app
```

Service-spezifische Konfiguration (z. B. hint-service)

Da es sich um ein Microservice-System handelt, besitzt jeder Service sein eigenes Konfigurations-Set innerhalb des Umgebungs-Overlays (z. B. envs/dev /hint-service). Die kustomization.yaml in diesem Verzeichnis orchestriert alle Anpassungen für diesen spezifischen Service.

Beispiel: envs/dev/hint-service/kustomization.yaml

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

# Hängt "-hint" an alle Ressourcennamen an, um Services zu unterscheiden
nameSuffix: "-hint"

# 1. Basis-Manifeste, die als Grundlage dienen

resources:
  - ../../base # Lädt die generischen Templates (Deployment, Service etc.)
  - admin-virtualservice.yaml
  - destination-rule.yaml

# 2. Wiederverwendbare Konfigurations-Komponenten

components:
  - ../../base/components/postgres  # Basis-Konfig für Postgres (z.B. env-Namen)
  - ../components/postgres          # Dev-spezifische Konfig für Postgres (z.B. URL)
  - ../../base/components/mongo
  - ../components/mongo
  - ../../base/components/kafka
  - ../components/kafka

# 3. Spezifisches Container-Image für diesen Service in dieser Umgebung

images:

  - name: app-image # Platzhalter-Name aus base/deployment.yaml
    newName: dev.docker.system.local/elpa-hint-tst/hint
    newTag: aee6dd3.18

# 4. Service-spezifische Umgebungsvariablen

configMapGenerator:
  - name: backend-cm
    behavior: merge
    literals:
      - SERVICE_NAME=hint-service
      - POSTGRES_SCHEMA_NAME=hin_develop

# 5. Gezielte Anpassungen an den Basis-Manifesten
patches:
  - path: virtualservice-patch.yaml
  - path: service-app-patch.yaml
  - path: deployment-patch.yaml
```

Erläuterung der Schlüssel:

- **resources**: Definiert die Basis-Manifeste, die angepasst werden sollen. In der Regel ist dies der Verweis auf das base-Verzeichnis.
- **components**: Importiert wiederverwendbare Konfigurationsbausteine. Hier wird das Schichtungsprinzip deutlich: Zuerst werden die Basis-Komponenten geladen, danach die umgebungsspezifischen Komponenten, die Werte überschreiben.
- **images**: Ersetzt den generischen Image-Namen aus der base-Konfiguration durch das konkrete Docker-Image (Name und Tag) für den Service. Dies ist besonders nützlich für CI/CD-Prozesse, da das Image-Tag einfach über den Befehl kustomize edit set image <image:tag> aktualisiert werden kann.
- **patches**: Definiert eine Liste von Patch-Dateien, die gezielte Änderungen an den geladenen resources vornehmen. Dies ist der Kern des Overlay-Prinzips. Ein Patch kann beispielsweise Ressourcen-Limits (cpu, memory) im Deployment anpassen, Routing-Regeln im VirtualService ändern oder Ports im Service-Objekt modifizieren, ohne die base-Dateien duplizieren zu müssen.

Das erste Service-Deployment

Wie wir aus dem vorherigen Abschnitt über Kubernetes und Kustomization wissen, gibt es beim Deployment eines Services in OCP4 viele Kubernetes-Manifest-Dateien und dementsprechend auch viele Deklarationen in der Kustomization-Datei. Die Initialisierung dieser Dateien ist bei allen Services nahezu identisch, sodass man einen Blueprint klonen und die notwendigen Werte umschreiben kann. Der Umschreibungsvorgang kann aufwendig und fehleranfällig sein, weshalb dieser Vorgang durch ein Skript (**init-service.sh**) automatisiert wird. Dieses Skript klont den Blueprint-Ordner, überschreibt die Werte-Platzhalter des Blueprints durch Benutzereingaben und legt den Deployment-Ordner des neuen Services im richtigen Umgebungsverzeichnis ab.

```
chmod +x init-service.sh
```

Danach müssen einige Eingaben als Antworten auf die Fragen des Skripts beantwortet werden.

```
In welcher Umgebung soll der Service bereitgestellt werden? (Bitte Option Nummer eingeben)
1) dev
2) abn
3) prod
#? 1
Sie haben die Umgebung 'dev' ausgewählt.

Wird PostgreSQL verwendet? (ja/nein - j/n): j
Wird Kafka verwendet? (ja/nein - j/n): j
Wird MongoDB verwendet? (ja/nein - j/n): n
Wird eine AUTH_URL benötigt? (ja/nein - j/n): j

Wie soll der neue Service heißen? (z.B. 'neuer-service'): beispiel-service
Wie lautet der Name des Docker-Images?: beispiel-image
Welchen Image-Tag soll das Image haben? (z.B. 'v1.0.0'): asd.c
Wie lautet der PostgreSQL Schema-Name? (leer lassen für Platzhalter): beispiel-schema
Bitte geben Sie die AUTH_URL ein (leer lassen für Platzhalter): beispiel.com
```

Nach der Ausführung wird eine allgemeine Information über das aktuell bereitgestellte Service-Set angezeigt.

elpa-elpa4/envs

```
 Der neue Service 'beispiel-service' wurde erfolgreich im Verzeichnis 'envs/dev/beispiel-service' erstellt.

Komponenten:
    - ../../base/components/postgres
    - ../components/postgres
    - ../../base/components/kafka
    - ../components/kafka

Konfiguration:
        - SERVICE_NAME=beispiel-service
        - POSTGRES_SCHEMA_NAME=beispiel-schema
        - AUTH_URL=beispiel.com

Hinweise:
  - Die Konfiguration wurde in 'envs/dev/beispiel-service/kustomization.yaml' erstellt
  - Überprüfen Sie die generierten Dateien vor dem Deployment
```

Nach der Generierung des Deployment-Sets muss noch ein wichtiger Schritt ausgeführt werden: die Anmeldung des Services in der Kustomization-Datei der entsprechenden Umgebung. Zum Beispiel: Wenn das Service-Deployment in der Dev-Umgebung stattfindet, muss das Deployment-Verzeichnis des Services als ein Element der Ressourcen der Dev-Kustomization definiert werden.

elpa-elpa4/envs

```
envs
 dev
    beispiel-service
       admin-virtualservice.yaml
       deployment-patch.yaml
       destination-rule.yaml
       kustomization.yaml
       service-app-patch.yaml
       virtualservice-patch.yaml
```

Am Ende, vor dem Commit und dem Mergen in den NOP-Branch zum Deployment, ist es sehr empfehlenswert, den `kustomize build`-Befehl einmal lokal auszuführen. Damit kann man mehr oder weniger sicherstellen, dass es keine Syntaxfehler im Deployment-Prozess gibt.

elpa-elpa4/envs

```
# Das Syntax von kustomization files bzw. auch andere Kubernets Objekt Files kann durch diese kommende Command geprüft werden

kubectl kustomize build envs/dev --enable-helm > output.yaml
```