🎓 Kubernetes Manifest Parameter - Deep Dive Guide

Dein vollständiges Nachschlagewerk für alle wichtigen Kubernetes-Parameter

---

📖 Inhaltsverzeichnis

---

1. Deployment Strategies

1.1 RollingUpdate vs. Recreate

Parameter	Was macht er?	Warum nutzen?	Anfänger-Empfehlung
`strategy.type: RollingUpdate`	Ersetzt Pods schrittweise: Erst neue Pods starten, dann alte beenden.	**Zero-Downtime Updates**: Deine App bleibt während des Updates erreichbar.	✅ **Standardwahl für 95% aller Fälle**
`strategy.type: Recreate`	Beendet **alle** alten Pods sofort, dann startet neue.	Für Apps, die **keine parallelen Versionen** dulden (z.B. Datenbankmigrationen).	⚠️ Nur wenn RollingUpdate Probleme macht

RollingUpdate Parameter im Detail:

```
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxUnavailable: 1      # Max. Anzahl Pods, die während Update "down" se
    maxSurge: 1            # Max. zusätzliche Pods über gewünschte Anzahl h
```

Parameter	Was macht er?	Warum wichtig?	Anfänger-Empfehlung
`maxUnavailable`	Wie viele Pods gleichzeitig down sein dürfen (Zahl oder %).	Kontrolliert Verfügbarkeit während Update. `0` = immer volle Kapazität.	`maxUnavailable: 1` (ein Pod darf offline sein)
`maxSurge`	Wie viele zusätzliche Pods temporär gestartet werden dürfen.	Schnellere Updates, aber mehr Ressourcen.	`maxSurge: 1` (ein zusätzlicher Pod)

Beispiel-Szenarien:

- **Webserver/API**: `RollingUpdate` mit `maxUnavailable: 0, maxSurge: 1` → keine Downtime
- **Datenbank**: `Recreate` → verhindert Konflikte zwischen alten/neuen Schemas
- **Job-Verarbeitung**: `RollingUpdate` mit `maxUnavailable: 25%` → schnellerer Rollout

---

2. Container Konfiguration

2.1 Image Pull Policy

Parameter	Was macht er?	Wann nutzen?	Anfänger-Empfehlung
`imagePullPolicy: Always`	Lädt Image **bei jedem Pod-Start** von Registry.	Entwicklung mit `:latest` Tag, um immer neueste Version zu haben.	✅ Für `:latest` Tags (aber besser: versionierte Tags!)
`imagePullPolicy: IfNotPresent`	Lädt nur, wenn Image lokal nicht existiert.	Produktion mit festen Versions-Tags (z.B. `v1.2.3`).	✅ **Standard für Produktion**
`imagePullPolicy: Never`	Nutzt nur lokal vorhandene Images, lädt nie.	Lokale Entwicklung mit Minikube/Docker Desktop.	⚠️ Nur für lokale Tests

Wichtig:
- Tag `:latest` triggert automatisch `Always` (auch wenn nicht angegeben)
- **Best Practice:** Nutze immer versionierte Tags wie `v1.2.3` statt `:latest`

```
# ❌ Nicht empfohlen (in Produktion)
image: myapp:latest

# ✅ Empfohlen
image: myapp:v1.2.3
imagePullPolicy: IfNotPresent
```

2.2 Image Pull Secrets

Parameter	Was macht er?	Warum wichtig?	Anfänger-Empfehlung
`imagePullSecrets`	Authentifizierung für private Container-Registries (Harbor, Docker Hub, etc.).	Ohne dieses Secret kann Kubernetes dein Image nicht pullen → `ImagePullBackOff` Fehler.	✅ **Pflicht für private Registries**

```
spec:
  imagePullSecrets:
    - name: harbor-regcred  # Name des Secrets mit Registry-Credentials
  containers:
    - name: app
      image: meine-registry.com/app:v1
```

2.3 Container Restart Policy

Parameter	Was macht er?	Wann nutzen?	Anfänger-Empfehlung
`restartPolicy: Always`	Pod wird automatisch neu gestartet (auch bei Erfolg).	Standard für langlebige Apps (Webserver, APIs).	✅ **Default für Deployments**
`restartPolicy: OnFailure`	Restart nur bei Fehler (Exit Code ≠ 0).	Jobs, die nur einmal laufen sollen.	✅ Für Jobs
`restartPolicy: Never`	Kein automatischer Restart.	One-Shot-Tasks, Debug-Pods.	⚠️ Selten nötig

---

3. Resource Management

KRITISCH für Cluster-Stabilität!

3.1 Requests vs. Limits

Parameter	Was macht er?	Warum lebenswichtig?	Anfänger-Empfehlung
`resources.requests`	**Garantierte** Ressourcen: Scheduler stellt sicher, dass Node genug hat.	Pod wird nur auf Node platziert, der genug freie Ressourcen hat.	✅ **IMMER setzen!**
`resources.limits`	**Maximum** an Ressourcen: Container wird bei Überschreitung gedrosselt/gekillt.	Verhindert, dass ein Container den ganzen Cluster lahmlegt.	✅ **IMMER setzen!**

```
resources:
  requests:      # "Ich brauche mindestens..."
    memory: "256Mi"
    cpu: "250m"
  limits:        # "Ich darf maximal..."
    memory: "512Mi"
    cpu: "500m"
```

3.2 CPU-Einheiten

Wert	Bedeutung	Wann nutzen?
`1` / `1000m`	1 voller CPU-Core	Rechenintensive Apps
`500m`	0.5 CPU-Core (50%)	Normale Webanwendungen
`100m`	0.1 CPU-Core (10%)	Leichtgewichtige Services

m = "Milli-CPU" (1000m = 1 Core)

3.3 Memory-Einheiten

Wert	Bedeutung	Wann nutzen?
`1Gi`	1 Gibibyte (1024^3 Bytes)	Datenbanken, Memory-intensive Apps
`512Mi`	512 Mebibyte	Standard Web-Apps
`128Mi`	128 Mebibyte	Sidecar-Container, kleine Services

Wichtig: `Mi` = Mebibyte (1024^2), `M` = Megabyte (1000^2) → bevorzuge `Mi` für Klarheit

3.4 Was passiert bei Überschreitung?

Ressource	Überschreitung von `requests`	Überschreitung von `limits`
CPU	Pod wird gedrosselt (throttled), läuft langsamer	Pod wird gedrosselt, **nie** gekillt
Memory	Pod läuft weiter (nutzt "Burst")	Pod wird **sofort gekillt** (OOMKilled)

Anfänger-Regel:

1. **Starte mit:** `requests = limits` (vorhersehbare Performance)
2. **Monitoring:** Schau dir echte Nutzung mit `kubectl top pods` an
3. **Optimiere:** Passe Werte nach echten Daten an

```
# ✅ Guter Start für kleine Web-App
resources:
  requests:
    memory: "256Mi"
    cpu: "100m"
  limits:
    memory: "512Mi"
    cpu: "500m"

# ✅ Datenbank (mehr Memory, stabiler)
resources:
  requests:
    memory: "1Gi"
    cpu: "500m"
  limits:
    memory: "2Gi"
    cpu: "1000m"
```

---

4. Health Checks (Probes)

Unterschied verstehen = kritisch für App-Stabilität!

4.1 Die 3 Probe-Typen

Probe	Was prüft sie?	Wann wird sie aktiv?	Bei Fehler?	Anfänger-Empfehlung
`startupProbe`	"Ist die App erstmals fertig gestartet?"	Nur beim **ersten Start**	Pod wird neu gestartet	✅ **Bei langsam startenden Apps** (>30s)
`livenessProbe`	"Läuft die App noch oder ist sie abgestürzt?"	**Dauerhaft** nach Start	Pod wird **neu gestartet**	✅ **Immer setzen!**
`readinessProbe`	"Ist die App bereit, Traffic zu empfangen?"	**Dauerhaft**	Traffic wird **gestoppt**, Pod bleibt laufen	✅ **Immer setzen!**

4.2 Probe-Mechanismen

Typ	Was passiert?	Wann nutzen?	Beispiel
`httpGet`	HTTP-Request an Endpoint	Web-Apps, APIs	`GET /health`
`tcpSocket`	TCP-Verbindung zu Port	Datenbanken, Redis	Port `5432`
`exec`	Shell-Command ausführen	Komplexe Checks	`pg_isready`

4.3 Probe-Parameter im Detail

```
livenessProbe:
  httpGet:
    path: /health        # Endpoint (erstelle einen in deiner App!)
    port: 8080
  initialDelaySeconds: 30  # Warte 30s nach Start, bevor erste Prüfung
  periodSeconds: 10        # Prüfe alle 10 Sekunden
  timeoutSeconds: 5        # Request darf max. 5s dauern
  failureThreshold: 3      # Nach 3 Fehlern → Pod neu starten
  successThreshold: 1      # Nach 1 Erfolg → OK
```

Parameter	Was macht er?	Typischer Wert	Warum wichtig?
`initialDelaySeconds`	Wartezeit vor erster Prüfung	10-60s	App braucht Zeit zum Starten
`periodSeconds`	Intervall zwischen Prüfungen	10s	Zu oft = Last, zu selten = langsame Fehlererkennung
`timeoutSeconds`	Max. Wartezeit pro Prüfung	1-5s	Überlastete App soll nicht als tot gelten
`failureThreshold`	Fehler bis Aktion (Restart/Traffic-Stop)	3	Verhindert Flapping bei kurzen Aussetzern
`successThreshold`	Erfolge bis "gesund"	1 (liveness), 2-3 (readiness)	Stabilität bei intermittierenden Problemen

4.4 Praxis-Beispiele

Beispiel 1: Web-API mit langsamer Initialisierung

```
containers:
  - name: api
    image: my-api:v1
    ports:
      - containerPort: 8080

    # Startup: Für Apps, die lange brauchen (DB-Verbindung, Cache-Warmup)
    startupProbe:
      httpGet:
        path: /health
        port: 8080
      initialDelaySeconds: 0
      periodSeconds: 5
      failureThreshold: 30       # 30 * 5s = 2.5 Minuten Zeit zum Starten

    # Liveness: Ist die App noch am Leben?
    livenessProbe:
      httpGet:
        path: /health            # Leichtgewichtiger Check!
        port: 8080
      periodSeconds: 10
      timeoutSeconds: 5
      failureThreshold: 3

    # Readiness: Kann die App Traffic verarbeiten?
    readinessProbe:
      httpGet:
        path: /ready             # Kann DB-Verbindung prüfen
        port: 8080
      periodSeconds: 5
      timeoutSeconds: 3
      failureThreshold: 2
```

Beispiel 2: PostgreSQL Datenbank

```
containers:
  - name: postgres
    image: postgres:15

    # Liveness: Ist Postgres-Prozess noch da?
    livenessProbe:
      exec:
        command:
          - /bin/sh
          - -c
          - pg_isready -U $POSTGRES_USER
      initialDelaySeconds: 30
      periodSeconds: 10
      failureThreshold: 3

    # Readiness: Kann Postgres Queries verarbeiten?
    readinessProbe:
      exec:
        command:
          - /bin/sh
          - -c
          - pg_isready -U $POSTGRES_USER
      initialDelaySeconds: 5
      periodSeconds: 5
      failureThreshold: 3
```

4.5 Typische Fehler vermeiden

❌ Fehler	✅ Lösung	Warum?
Kein `/health` Endpoint	Erstelle einen in deiner App	Probes brauchen etwas zum Prüfen
`initialDelaySeconds` zu kurz	Messe echte Startzeit deiner App	Sonst Restart-Loop
Liveness = schwerer Check (DB-Query)	Liveness = leicht, Readiness = schwer	Liveness killt Pod → nur bei echtem Crash
`failureThreshold: 1`	Mindestens `3`	Verhindert Flapping bei Netzwerk-Hickups
Readiness fehlt	Immer setzen!	Sonst bekommt Pod Traffic, bevor er bereit ist

Anfänger-Faustregel:
1. **Erstelle** `/health` **Endpoint** in deiner App (antwortet `200 OK`)
2. **StartupProbe:** Nur bei Apps mit >30s Startzeit
3. **LivenessProbe:** Immer! Prüft, ob App "lebt"
4. **ReadinessProbe:** Immer! Prüft, ob App Traffic verarbeiten kann

---

5. Security Context

Sicherheit von Anfang an!

5.1 Pod Security Context

Parameter	Was macht er?	Warum wichtig?	Anfänger-Empfehlung
`runAsNonRoot: true`	Erzwingt, dass Container **nicht als root** läuft	Verhindert Privilege-Escalation-Angriffe	✅ **IMMER aktivieren**
`runAsUser: 1000`	Setzt User-ID explizit	Vorhersehbare Permissions	✅ Bei Permission-Problemen
`fsGroup: 1000`	Setzt Group-ID für Volumes	Volumes sind für User lesbar/schreibbar	✅ Bei Volume-Permission-Fehlern

```
spec:
  securityContext:
    runAsNonRoot: true      # ✅ Niemals als root!
    runAsUser: 1000         # User-ID
    fsGroup: 1000           # Group für Volumes
```

5.2 Container Security Context

Parameter	Was macht er?	Warum nutzen?	Anfänger-Empfehlung
`readOnlyRootFilesystem: true`	Container kann nur in `/tmp` und Volumes schreiben	Verhindert Malware-Installation im Container	✅ Wenn App es unterstützt
`allowPrivilegeEscalation: false`	Verhindert, dass Prozess mehr Rechte bekommt	Schließt kritische Sicherheitslücke	✅ **IMMER setzen**
`capabilities.drop: [ALL]`	Entfernt alle Linux-Capabilities	Minimale Rechte = minimales Risiko	✅ **Best Practice**

```
containers:
  - name: app
    image: my-app:v1
    securityContext:
      allowPrivilegeEscalation: false   # ✅ Keine Rechte-Erhöhung
      runAsNonRoot: true                # ✅ Nicht als root
      readOnlyRootFilesystem: true      # ✅ Filesystem read-only
      capabilities:
        drop:
          - ALL                         # ✅ Alle Capabilities entfernen
```

5.3 Praxis-Beispiel: Sichere Web-App

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: secure-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: secure-app
  template:
    metadata:
      labels:
        app: secure-app
    spec:
      # Pod-Level Security
      securityContext:
        runAsNonRoot: true
        runAsUser: 1000
        fsGroup: 1000

      containers:
        - name: app
          image: my-secure-app:v1

          # Container-Level Security
          securityContext:
            allowPrivilegeEscalation: false
            readOnlyRootFilesystem: true
            capabilities:
              drop:
                - ALL

          # Wenn App in /tmp schreiben muss
          volumeMounts:
            - name: tmp
              mountPath: /tmp

      volumes:
```

```
        - name: tmp
          emptyDir: {}   # Temporäres Volume für /tmp
```

5.4 Häufige Permission-Probleme lösen

Problem: `Permission denied` beim Zugriff auf Volume

```
# ✅ Lösung:
spec:
  securityContext:
    fsGroup: 999    # Postgres nutzt User 999
  containers:
    - name: postgres
      image: postgres:15
      securityContext:
        runAsUser: 999
```

Problem: App braucht Schreibrechte im Container

```
# ✅ Lösung: Nutze Volumes statt readOnlyRootFilesystem
containers:
  - name: app
    securityContext:
      readOnlyRootFilesystem: true
    volumeMounts:
      - name: temp-data
        mountPath: /app/temp
volumes:
  - name: temp-data
    emptyDir: {}
```

---

6. Volumes & Persistent Storage

6.1 Volume-Typen im Überblick

Typ	Lebensdauer	Wann nutzen?	Anfänger-Empfehlung
`emptyDir`	Pod-Lifetime (gelöscht bei Pod-Neustart)	Temporäre Daten, Cache	✅ Für nicht-kritische Daten
`hostPath`	Auf Node-Disk, überlebt Pod-Neustarts	Lokales Testing (Minikube)	⚠️ **NUR für Entwicklung!**
`persistentVolumeClaim`	Persistent, überlebt Pod/Node-Neustarts	Datenbanken, User-Uploads	✅ **Für Produktion!**
`configMap`	Konfigurationsdateien als Volume	Config-Files in Container mounten	✅ Für Configs
`secret`	Sensible Daten als Volume	Certificates, Keys	✅ Für Secrets

6.2 emptyDir - Temporärer Storage

```
volumes:
  - name: cache
    emptyDir: {}           # Standard: Disk

  - name: fast-cache
    emptyDir:
      medium: Memory       # In RAM (schnell, aber begrenzt!)
      sizeLimit: 100Mi     # Max. 100MB
```

Parameter	Was macht er?	Wann nutzen?
`emptyDir: {}`	Temporäres Verzeichnis auf Disk	Logs, temp files, Cache
`medium: Memory`	Nutzt RAM statt Disk	Performance-kritische Caches
`sizeLimit`	Max. Größe	RAM-Schutz, Resource-Limits

6.3 PersistentVolumeClaim (PVC)

Für alle Daten, die erhalten bleiben müssen!

```
# PVC erstellen
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-data
  annotations:
    argocd.argoproj.io/sync-options: Prune=false  # ✅ ArgoCD soll nicht lö
spec:
  accessModes:
    - ReadWriteOnce      # Nur 1 Pod kann schreiben
  resources:
    requests:
      storage: 10Gi      # 10 GB Storage
  storageClassName: standard  # Abhängig vom Cluster
```

Parameter	Was macht er?	Optionen	Anfänger-Empfehlung
`accessModes`	Wer darf wie zugreifen?	Siehe unten	`ReadWriteOnce` (Standard)
`storage`	Wie viel Speicher?	`1Gi` , `10Gi` , `100Gi`	Start mit kleinem Wert, erweitern bei Bedarf
`storageClassName`	Welche Storage-Klasse?	`standard` , `fast-ssd` , etc.	Nutze Cluster-Default

Access Modes im Detail

Mode	Abkürzung	Was bedeutet es?	Wann nutzen?
ReadWriteOnce	RWO	1 Pod (auf 1 Node) kann lesen+schreiben	Datenbanken, Apps mit 1 Replica
ReadOnlyMany	ROX	Viele Pods können lesen	Shared Config-Files
ReadWriteMany	RWX	Viele Pods können lesen+schreiben	Shared Storage (NFS), Multi-Pod-Apps

Wichtig: Nicht alle Storage-Klassen unterstützen alle Modes!

6.4 PVC in StatefulSet nutzen

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: postgres
spec:
  serviceName: postgres
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    spec:
      containers:
        - name: postgres
          image: postgres:15
          volumeMounts:
            - name: data
              mountPath: /var/lib/postgresql/data

  # volumeClaimTemplates: Automatisch PVC pro Replica erstellen
  volumeClaimTemplates:
    - metadata:
        name: data
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 10Gi
```

6.5 Reclaim Policy - Was passiert bei Löschung?

Wird im PersistentVolume (PV) definiert, nicht im PVC:

Policy	Was passiert bei PVC-Löschung?	Wann nutzen?
Retain	Daten bleiben erhalten, manuelles Cleanup nötig	Produktion (Datensicherheit)
Delete	PV und Daten werden gelöscht	Development
Recycle	Veraltet, nicht nutzen	-

```
# PV mit Retain-Policy
apiVersion: v1
kind: PersistentVolume
metadata:
  name: postgres-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain  # ✅ Daten bleiben!
  storageClassName: standard
```

---

7. Networking (Services)

Services machen Pods erreichbar

7.1 Service-Typen

Typ	Was macht er?	Erreichbar von?	Wann nutzen?	Anfänger-Empfehlung
ClusterIP	Interne IP, nur im Cluster	Nur innerhalb des Clusters	Backend-Services, Datenbanken	✅ **Standard**
NodePort	Port auf allen Nodes	Von außen über `<NodeIP>:<NodePort>`	Lokales Testing, Development	✅ Für lokale Tests
LoadBalancer	Externer Load Balancer	Internet (öffentliche IP)	Cloud-Umgebungen (AWS, GCP, Azure)	✅ In Cloud-Prod
ExternalName	DNS-Alias	-	Externe Services (z.B. AWS RDS)	⚠️ Fortgeschritten

7.2 ClusterIP (Standard)

```
apiVersion: v1
kind: Service
metadata:
  name: postgres-service
spec:
  type: ClusterIP        # Standard (kann auch weggelassen werden)
  selector:
    app: postgres        # Welche Pods gehören zu diesem Service?
  ports:
    - name: postgres
      port: 5432         # Port des Services (wie andere ihn ansprechen)
      targetPort: 5432   # Port am Container
      protocol: TCP
```

Parameter	Was macht er?	Warum wichtig?
`selector`	Label-Matching zu Pods	Definiert, welche Pods Traffic bekommen
`port`	Service-Port	Port, über den andere den Service ansprechen
`targetPort`	Container-Port	Port, auf dem Container lauscht

DNS: Service ist erreichbar über `postgres-service.development.svc.cluster.local` (oder kurz: `postgres-service`)

7.3 NodePort (für lokale Entwicklung)

```
apiVersion: v1
kind: Service
metadata:
  name: harbor
spec:
  type: NodePort
  selector:
    app: harbor
  ports:
    - port: 80          # ClusterIP Port
      targetPort: 8080  # Container Port
      nodePort: 30002   # ✅ Port auf Node (30000-32767)
```

Parameter	Wert	Bedeutung
`nodePort`	30000-32767	Fester Port auf allen Nodes

Erreichbar über: `http://localhost:30002` (bei Minikube/Docker Desktop)

7.4 LoadBalancer (Cloud)

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
spec:
  type: LoadBalancer    # ✅ Cloud-Provider erstellt Load Balancer
  selector:
    app: frontend
  ports:
    - port: 80
      targetPort: 8080
```

Cloud-Provider (AWS/GCP/Azure) erstellt automatisch externen Load Balancer mit öffentlicher IP.

7.5 Headless Service (für StatefulSets)

```
apiVersion: v1
kind: Service
metadata:
  name: postgres
spec:
  clusterIP: None        # ✅ Headless = keine ClusterIP
  selector:
    app: postgres
  ports:
    - port: 5432
```

Wann nutzen?
- StatefulSets, wo jeder Pod direkt ansprechbar sein soll
- Service Discovery per DNS: `postgres-0.postgres.development.svc.cluster.local`

---

8. ConfigMaps & Secrets

Konfiguration von Code trennen!

8.1 ConfigMap vs. Secret

	ConfigMap	Secret
Für	Nicht-sensible Configs	Passwörter, Tokens, Keys
Kodierung	Klartext	Base64 (⚠️ nicht verschlüsselt!)
Wann nutzen?	App-Settings, Feature-Flags	DB-Passwörter, API-Keys

8.2 ConfigMap erstellen

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  # Key-Value Pairs
  DATABASE_HOST: postgres-service
  DATABASE_NAME: studentsdb
  LOG_LEVEL: "info"

  # Oder ganze Dateien
  app.properties: |
    server.port=8080
    spring.datasource.url=jdbc:postgresql://postgres:5432/db
```

8.3 ConfigMap nutzen: Als Environment Variables

```
containers:
  - name: app
    image: my-app:v1
    env:
      # ✅ Einzelne Werte
      - name: DATABASE_HOST
        valueFrom:
          configMapKeyRef:
            name: app-config
            key: DATABASE_HOST

    # ✅ ALLE Werte auf einmal
    envFrom:
      - configMapRef:
          name: app-config
```

8.4 ConfigMap nutzen: Als Volume (Datei)

```
containers:
  - name: app
    volumeMounts:
      - name: config
        mountPath: /etc/config
        readOnly: true
volumes:
  - name: config
    configMap:
      name: app-config
```

Ergebnis: `/etc/config/app.properties` enthält Config-File

8.5 Secret erstellen

```
apiVersion: v1
kind: Secret
metadata:
  name: db-credentials
type: Opaque
data:
  username: YXBwX3VzZXI=        # Base64: app_user
  password: bXlwYXNzd29yZDEyMw==  # Base64: mypassword123
```

Base64 kodieren:

```
[Convert]::ToBase64String([Text.Encoding]::UTF8.GetBytes("app_user"))
```

8.6 Secret nutzen

```
containers:
  - name: app
    env:
      # ✅ Als Environment Variable
      - name: DB_USER
        valueFrom:
          secretKeyRef:
            name: db-credentials
            key: username

      - name: DB_PASSWORD
        valueFrom:
          secretKeyRef:
            name: db-credentials
            key: password
```

8.7 SealedSecret (für GitOps)

Problem: Normale Secrets sind nur Base64, nicht verschlüsselt!

Lösung: SealedSecrets (verschlüsselt mit Public-Key)

```
apiVersion: bitnami.com/v1alpha1
kind: SealedSecret
metadata:
  name: db-credentials
spec:
  encryptedData:
    username: AgBH8j2... # ✅ Echt verschlüsselt!
    password: AgCK9d...
```

Erstellen:

```
kubeseal --format yaml < db-credentials-secret.yaml > db-credentials-sealed
```

---

9. Init Containers

Führen Vorbereitungs-Tasks aus, bevor Haupt-Container startet

9.1 Was sind Init Containers?

Eigenschaft	Init Container	Haupt-Container
Wann startet?	Vor Haupt-Container	Nach Init-Containern
Laufen parallel?	Nein (sequenziell)	Ja
Bei Fehler?	Pod startet nicht	Pod wird neu gestartet
Wann nutzen?	Setup-Tasks	Eigentliche App

9.2 Typische Use Cases

1. **Warten auf Abhängigkeiten** (z.B. Datenbank)
2. **Daten vorbereiten** (z.B. Git-Repo klonen)
3. **Permissions setzen** (z.B. `chown` auf Volume)
4. **Config generieren**

9.3 Beispiel: Warten auf Datenbank

```
spec:
  # Init Container läuft ZUERST
  initContainers:
    - name: wait-for-postgres
      image: busybox:1.36
      command:
        - sh
        - -c
        - |
          echo "Waiting for postgres..."
          until nc -z postgres-service 5432; do
            echo "Postgres not ready, waiting..."
            sleep 2
          done
          echo "Postgres is ready!"

  # Haupt-Container startet erst, wenn Init fertig ist
  containers:
    - name: app
      image: my-app:v1
      env:
        - name: DATABASE_HOST
          value: postgres-service
```

9.4 Beispiel: Volume Permissions

```
spec:
  initContainers:
    - name: fix-permissions
      image: busybox:1.36
      command:
        - sh
        - -c
        - chown -R 999:999 /var/lib/postgresql/data
      volumeMounts:
        - name: data
          mountPath: /var/lib/postgresql/data
      securityContext:
        runAsUser: 0  # Muss als root laufen für chown

  containers:
    - name: postgres
      image: postgres:15
      volumeMounts:
        - name: data
          mountPath: /var/lib/postgresql/data
```

9.5 Mehrere Init Container (sequenziell)

```
spec:
  initContainers:
    # 1. Init Container
    - name: clone-repo
      image: git:2.40
      command: ['git', 'clone', 'https://...', '/data']
      volumeMounts:
        - name: data
          mountPath: /data

    # 2. Init Container (läuft nach clone-repo)
    - name: install-deps
      image: node:18
      command: ['npm', 'install']
      workingDir: /data
      volumeMounts:
        - name: data
          mountPath: /data

  # Haupt-Container startet als Letztes
  containers:
    - name: app
      image: node:18
      command: ['npm', 'start']
      volumeMounts:
        - name: data
          mountPath: /app
```

Reihenfolge: `clone-repo` → `install-deps` → `app`

---

10. StatefulSet Parameter

Für stateful Apps (Datenbanken, Message Queues)

10.1 StatefulSet vs. Deployment

	Deployment	StatefulSet
Pod-Namen	Zufällig (z.B. `app-abc123-xyz`)	Stabil (z.B. `postgres-0`, `postgres-1`)
Reihenfolge	Parallel	Sequenziell (0 → 1 → 2)
Storage	Shared oder keins	Eigenes PVC pro Pod
DNS	Service-ClusterIP	Jeder Pod: `pod-name.service.ns.svc`
Wann nutzen?	Stateless Apps	Datenbanken, Zookeeper, Kafka

10.2 StatefulSet Besonderheiten

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: postgres
spec:
  serviceName: postgres   # ✅ Headless Service (clusterIP: None)
  replicas: 3
  selector:
    matchLabels:
      app: postgres

  # Pod-Template (wie bei Deployment)
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
        - name: postgres
          image: postgres:15
          volumeMounts:
            - name: data
              mountPath: /var/lib/postgresql/data

  # ✅ volumeClaimTemplates: Erstellt PVC pro Pod
  volumeClaimTemplates:
    - metadata:
        name: data
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 10Gi
```

Parameter	Was macht er?	Warum wichtig?
`serviceName`	Referenz zu Headless Service	DNS: `postgres-0.postgres`
`volumeClaimTemplates`	PVC-Template pro Replica	Jeder Pod bekommt eigenen Storage

10.3 Update-Strategien

Strategie	Was passiert?	Wann nutzen?
`RollingUpdate` (default)	Sequenziell von höchster zu niedrigster Ordinal (2→1→0)	Standard
`OnDelete`	Manueller Update (Pod muss gelöscht werden)	Volle Kontrolle gewünscht

```
spec:
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      partition: 2  # ✅ Nur Pods >= 2 updaten (Canary Deployment)
```

10.4 Partition (Canary Deployments)

```
spec:
  replicas: 5
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      partition: 3  # Nur postgres-3 und postgres-4 werden geupdatet
```

Use Case: Teste neue Version auf 2 Pods, Rest bleibt auf alter Version

---

11. Job & CronJob Parameter

11.1 Job (einmalige Tasks)

```
apiVersion: batch/v1
kind: Job
metadata:
  name: database-migration
spec:
  template:
    spec:
      containers:
        - name: migrate
          image: migrate-tool:v1
          command: ['./migrate.sh']
      restartPolicy: OnFailure  # ✅ Job-spezifisch!

  # Job-Parameter
  backoffLimit: 4               # Max. 4 Versuche bei Fehler
  activeDeadlineSeconds: 600    # Max. 10 Minuten Laufzeit
  ttlSecondsAfterFinished: 3600 # Job nach 1h löschen
```

Parameter	Was macht er?	Typischer Wert
`backoffLimit`	Restart-Versuche bei Fehler	3-6
`activeDeadlineSeconds`	Max. Laufzeit (timeout)	600-3600s
`ttlSecondsAfterFinished`	Auto-Cleanup nach Erfolg	3600-86400s

11.2 CronJob (geplante Tasks)

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: backup-cronjob
spec:
  schedule: "0 2 * * *"         # ✅ Jeden Tag um 2 Uhr nachts
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: backup
              image: backup-tool:v1
              command: ['./backup.sh']
          restartPolicy: OnFailure

  # CronJob-Parameter
  successfulJobsHistoryLimit: 3  # Behalte 3 erfolgreiche Jobs
  failedJobsHistoryLimit: 1      # Behalte 1 fehlgeschlagenen Job
  concurrencyPolicy: Forbid      # Kein paralleles Laufen
```

11.3 Cron Schedule Format

```
┌──────────────── Minute (0-59)
│ ┌────────────── Stunde (0-23)
│ │ ┌──────────── Tag des Monats (1-31)
│ │ │ ┌────────── Monat (1-12)
│ │ │ │ ┌──────── Wochentag (0-6, Sonntag=0)
│ │ │ │ │
* * * * *
```

Schedule	Bedeutung
`*/5 * * * *`	Alle 5 Minuten
`0 * * * *`	Jede Stunde (zur vollen Stunde)
`0 2 * * *`	Täglich um 2 Uhr nachts
`0 0 * * 0`	Jeden Sonntag um Mitternacht
`0 0 1 * *`	Ersten Tag des Monats

11.4 Concurrency Policy

Policy	Was passiert bei Überlappung?	Wann nutzen?
`Allow`	Jobs laufen parallel	Unabhängige Tasks
`Forbid`	Neuer Job wird übersprungen	Nur 1 Job gleichzeitig erlaubt
`Replace`	Alter Job wird gekillt	Immer neueste Version laufen lassen

---

12. Ingress Parameter

Ingress = Eintrittspunkt von außen (HTTP/HTTPS)

12.1 Basis-Ingress

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app-ingress
  annotations:
    kubernetes.io/ingress.class: nginx  # Ingress Controller
spec:
  rules:
    - host: myapp.example.com           # Domain
      http:
        paths:
          - path: /                     # URL-Pfad
            pathType: Prefix            # Matching-Strategie
            backend:
              service:
                name: frontend-service  # Ziel-Service
                port:
                  number: 80
```

12.2 Path Types

PathType	Matching-Verhalten	Beispiel
`Prefix`	Alle Pfade, die mit `/api` starten	`/api/users`, `/api/products`
`Exact`	Nur exakter Match	Nur `/api` (nicht `/api/users`)
`ImplementationSpecific`	Abhängig vom Ingress Controller	-

12.3 Multiple Hosts & Paths

```
spec:
  rules:
    # Host 1
    - host: api.example.com
      http:
        paths:
          - path: /v1
            pathType: Prefix
            backend:
              service:
                name: api-v1
                port:
                  number: 8080

          - path: /v2
            pathType: Prefix
            backend:
              service:
                name: api-v2
                port:
                  number: 8080

    # Host 2
    - host: admin.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: admin-panel
                port:
                  number: 80
```

12.4 TLS/HTTPS

```
spec:
  tls:
    - hosts:
        - myapp.example.com
      secretName: tls-secret  # Secret mit cert + key
  rules:
    - host: myapp.example.com
      # ...
```

TLS Secret erstellen:

```
kubectl create secret tls tls-secret \
  --cert=path/to/cert.pem \
  --key=path/to/key.pem
```

12.5 Nützliche Annotations (NGINX Ingress)

```
metadata:
  annotations:
    # Timeouts
    nginx.ingress.kubernetes.io/proxy-connect-timeout: "60"
    nginx.ingress.kubernetes.io/proxy-send-timeout: "60"
    nginx.ingress.kubernetes.io/proxy-read-timeout: "60"

    # Body Size (für File-Uploads)
    nginx.ingress.kubernetes.io/proxy-body-size: "50m"

    # HTTPS Redirect
    nginx.ingress.kubernetes.io/force-ssl-redirect: "true"

    # CORS
    nginx.ingress.kubernetes.io/enable-cors: "true"
    nginx.ingress.kubernetes.io/cors-allow-origin: "*"
```

---

13. Pod Disruption Budget

Verhindert zu viele gleichzeitige Pod-Ausfälle (bei Updates, Node-Wartung)

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: app-pdb
spec:
  minAvailable: 2          # Min. 2 Pods müssen laufen
  # ODER:
  # maxUnavailable: 1      # Max. 1 Pod darf down sein
  selector:
    matchLabels:
      app: frontend
```

Parameter	Was macht er?	Wann nutzen?
`minAvailable`	Mindestanzahl verfügbarer Pods	Absolute Zahl wichtig
`maxUnavailable`	Max. gleichzeitig nicht-verfügbare Pods	Prozentual sinnvoll

Use Case: Bei `replicas: 3` und `minAvailable: 2` kann Cluster-Admin nur 1 Node gleichzeitig warten.

---

14. Affinity & Anti-Affinity

Steuert, auf welchen Nodes Pods landen

14.1 Node Affinity (Pod zu Node)

```
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:  # MUSS
        nodeSelectorTerms:
          - matchExpressions:
              - key: disktype
                operator: In
                values:
                  - ssd
```

Use Case: GPU-Workloads müssen auf GPU-Nodes

14.2 Pod Anti-Affinity (Pods verteilen)

```
spec:
  affinity:
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:  # SOLLTE
        - weight: 100
          podAffinityTerm:
            labelSelector:
              matchLabels:
                app: frontend
            topologyKey: kubernetes.io/hostname
```

Use Case: Verteile Replicas auf verschiedene Nodes (High Availability)

---

15. Namespace & Labels

15.1 Namespaces

Logische Trennung von Ressourcen

```
apiVersion: v1
kind: Namespace
metadata:
  name: development
```

Use Cases:
- **Environments:** `development`, `staging`, `production`
- **Teams:** `team-frontend`, `team-backend`
- **Mandanten:** `customer-a`, `customer-b`

15.2 Labels

Key-Value-Metadaten für Ressourcen

```
metadata:
  labels:
    app: frontend           # Applikations-Name
    version: v1.2.3         # Version
    tier: frontend          # Layer (frontend/backend/database)
    environment: production # Environment
```

Best Practices:
- `app`: Applikations-Name
- `version`: Version
- `tier`: Frontend/Backend/Database
- `environment`: prod/staging/dev

Nutzen von Labels:

```
# Alle Frontend-Pods
kubectl get pods -l app=frontend

# Nur Production
kubectl get pods -l environment=production

# Kombiniert
kubectl get pods -l app=frontend,environment=production
```

🎯 Cheat Sheet: Die 10 wichtigsten Regeln für Anfänger

#	Regel	Warum?
1️⃣	**Immer `resources` setzen** (requests + limits)	Cluster-Stabilität
2️⃣	**Immer Health Probes** (liveness + readiness)	Auto-Healing
3️⃣	`runAsNonRoot: true` in Security Context	Sicherheit
4️⃣	**Versionierte Image-Tags** (nicht `:latest`)	Reproduzierbare Deployments
5️⃣	**Secrets für sensible Daten**, ConfigMaps für Rest	Trennung
6️⃣	`imagePullPolicy: IfNotPresent` in Produktion	Performance
7️⃣	**PVC mit `Prune=false` Annotation**	Datenschutz
8️⃣	**Labels konsequent nutzen**	Organisation
9️⃣	**RollingUpdate-Strategy**	Zero-Downtime
🔟	**Init Containers für Dependencies**	Sauberer Start

📚 Weiterführende Ressourcen

- **Offizielle Kubernetes Docs:** https://kubernetes.io/docs/
- **kubectl Cheat Sheet:** https://kubernetes.io/docs/reference/kubectl/cheatsheet/

- **Best Practices:** https://kubernetes.io/docs/concepts/configuration/overview/

Dieses Dokument wurde erstellt als Nachschlagewerk für dein Manifest-Projekt.
Stand: 2026-02-03