Kubernetes Deployment Blueprint - Komplette Dokumentation

Generiert am: 2026-02-03 13:27:13

==========

GETTING-STARTED.md

==========

🎓 Student API - Kubernetes Deployment Guide für Einsteiger

> **Von 0 auf Production-Ready in 15 Minuten**
> Ein vollständiger Blueprint für absolute Kubernetes-Anfänger

---

👋 Willkommen!

Du hast gerade deine erste ASP.NET Core API gebaut und möchtest sie in Kubernetes deployen?
Perfekt! Diese Anleitung führt dich Schritt für Schritt durch den gesamten Prozess.

Was du hier findest:

✅ **Einfache Erklärungen** (kein Fachchinesisch)
✅ **Klick-für-Klick Anleitungen** (für Harbor & ArgoCD)
✅ **Kommentierte YAML-Dateien** (jede Zeile erklärt)
✅ **Troubleshooting-Guide** (für häufige Fehler)
✅ **Automatische Backups** (Datenverlust vermeiden)

🗂️ Dokumentation (Wo anfangen?)

🚀 Neu hier? START HIER:

1️⃣ QUICK-START-GUIDE.md

Dauer: 15 Minuten
Inhalt:
- ✅ Voraussetzungen prüfen
- ✅ Schritt-für-Schritt Deployment (10 Schritte)
- ✅ Harbor Setup (mit Screenshots-Beschreibung)
- ✅ ArgoCD Setup (Klick-für-Klick)
- ✅ Erste Student anlegen & testen

👉 **Perfekt für:** "Ich will einfach, dass es funktioniert!"

📚 Tieferes Verständnis?

2️⃣ DEPLOYMENT-BLUEPRINT.md

Dauer: 30-45 Minuten Lesen
Inhalt:
- 📦 Projekt-Übersicht
- 📁 Ordnerstruktur erklärt
- 🎓 Wichtige Begriffe (Pod, Service, Ingress, etc.)
- 🎯 Ausführlicher Workflow (mit Erklärungen)
- 🔐 Harbor & ImagePullSecret detailliert
- 🔄 ArgoCD Integration & GitOps
- 🛠️ Troubleshooting (alle Fehler-Szenarien)
- 📝 Daily Workflow (Code-Änderungen deployen)

👉 **Perfekt für:** "Ich will verstehen, was da passiert!"

3️⃣ YAML-EXAMPLES.md

Dauer: 1 Stunde (Referenz-Dokument)
Inhalt:
- 📝 Jede YAML-Datei vollständig kommentiert
- 🔍 Jede Zeile erklärt (was macht das?)
- 🎨 Visuelle Diagramme (wie hängt alles zusammen?)
- 💡 Quick Reference (häufige Patterns)
- 📖 Namespace, ConfigMap, Secret, Deployment, Service, Ingress, PVC

👉 **Perfekt für:** "Ich will YAML schreiben lernen!"

4️⃣ ARCHITECTURE.md

Dauer: 20 Minuten (optional)
Inhalt:
- 🏗️ System-Architektur Diagramm
- 🔄 Datenfluss (Browser → Datenbank)
- 🔐 Security Layers
- 💾 Backup & Recovery Architektur
- 🏗️ Image Build Pipeline
- 🔄 GitOps Workflow (visuell)
- 📈 Skalierungs-Szenarien

👉 **Perfekt für:** "Ich will das große Bild sehen!"

📦 Was ist diese App?

Student API - REST Service für Studenten-Verwaltung

Funktionen:
- ✅ Studenten anlegen (POST `/api/student`)
- ✅ Studenten auflisten (GET `/api/student`)
- ✅ Studenten löschen (DELETE `/api/student/{id}`)
- ✅ Swagger UI (interaktive Dokumentation)

Tech Stack:
- **Backend:** ASP.NET Core 8.0 (C#)
- **Datenbank:** PostgreSQL 15
- **ORM:** Entity Framework Core
- **API-Doc:** Swagger/OpenAPI
- **Container:** Docker
- **Orchestrierung:** Kubernetes
- **GitOps:** ArgoCD
- **Registry:** Harbor

Erreichbar unter:

```
http://localhost/swagger       (Swagger UI)
http://localhost/api/student   (API Endpoint)
```

🎯 Lernziele

Nach diesem Guide kannst du:

- ✅ **Docker Images bauen** und zu Harbor pushen
- ✅ **Kubernetes Manifeste** verstehen und schreiben
- ✅ **Secrets & ConfigMaps** korrekt nutzen
- ✅ **Services & Ingress** konfigurieren
- ✅ **ArgoCD** für GitOps einsetzen
- ✅ **Persistenten Speicher** (PVCs) verwalten
- ✅ **Backups** erstellen und wiederherstellen
- ✅ **Fehler debuggen** (Logs, Events, Describe)
- ✅ **Deployments aktualisieren** (Rolling Updates)
- ✅ **Zero-Downtime Deployments** erreichen

🛠️ Voraussetzungen (5 Min Setup)

✅ Must-Have (lokal installiert):

Tool	Version	Zweck	Installation
Docker Desktop	4.25+	Container-Runtime + Kubernetes	Download
Kubernetes	1.28+	Orchestrierung (in Docker Desktop aktivieren)	Settings → Kubernetes → ✓ Enable
kubectl	1.28+	Kubernetes CLI (kommt mit Docker Desktop)	`kubectl version`
Harbor	2.9+	Private Registry	Läuft auf `localhost:30002`
ArgoCD	2.9+	GitOps Tool	Installation siehe unten

✅ Optional (aber empfohlen):

Tool	Zweck	Installation
Git	Version Control	Download
PowerShell 7+	Moderne Shell	Download
VS Code	Editor	Download
Postman	API-Testing (Alternative zu Swagger)	Download

🔧 Schnell-Check:

```
# Alle Tools installiert?
docker version          # Sollte Client + Server zeigen
kubectl version --client # Sollte Version zeigen
kubectl get nodes       # Sollte: docker-desktop   Ready

# Harbor läuft?
# Browser: http://localhost:30002 (Login-Seite sichtbar?)

# ArgoCD installiert?
kubectl get pods -n argocd  # Sollte mehrere Pods zeigen
```

Alle ✅? → Weiter zum Quick-Start!
Fehler? → Siehe Voraussetzungen-Setup unten

🚀 Schnellstart (15 Min)

Option A: Ich will sofort starten!

```
# 1. Repo klonen (falls noch nicht)
cd C:\Users\hedin\source\repos\manifest\manifest\manifest

# 2. Image bauen & pushen
.\pipeline.ps1

# 3. Namespace & Secret erstellen
kubectl create namespace development

kubectl create secret docker-registry harbor-regcred `
  --docker-server=localhost:30002 `
  --docker-username=admin `
  --docker-password=Harbor12345 `
  --docker-email=admin@local `
  --namespace=development

# 4. ArgoCD Application deployen
kubectl apply -f argocd/application.yaml

# 5. Warten (2-3 Min)
kubectl get pods -n development -w

# 6. Browser öffnen
Start http://localhost/swagger
```

Funktioniert nicht? → Siehe QUICK-START-GUIDE.md für detaillierte Schritte

---

Option B: Ich will verstehen, was passiert!

📖 **Lies erst:** DEPLOYMENT-BLUEPRINT.md
Dann folge den Schritten dort.

📁 Projekt-Struktur (Überblick)

```
manifest/
│
├── 📚 DOKUMENTATION (NEU!)
│   ├── GETTING-STARTED.md          ← DU BIST HIER
│   ├── QUICK-START-GUIDE.md        ← 15-Min Schnellstart
│   ├── DEPLOYMENT-BLUEPRINT.md     ← Vollständiger Guide
│   ├── YAML-EXAMPLES.md            ← Alle YAMLs erklärt
│   └── ARCHITECTURE.md             ← Architektur-Diagramme
│
├── 🛠️ APP-CODE
│   ├── StudentApi/                 ← ASP.NET Core App
│   │   ├── Controllers/
│   │   ├── Models/
│   │   ├── Data/
│   │   ├── Dockerfile              ← Image-Bauplan
│   │   └── Program.cs
│   └── docker-compose.yml          ← Lokale Entwicklung
│
├── ☸️ KUBERNETES
│   ├── k8s/                        ← Alle Kubernetes-Manifeste
│   │   ├── Student-api/
│   │   │   ├── Namespace.yaml      ← Erstellt "development" Namespace
│   │   │   ├── ConfigMap.yaml      ← Nicht-geheime Einstellungen
│   │   │   ├── Deployment.yaml     ← Hauptdatei (App + Pods)
│   │   │   └── Ingress.yaml        ← HTTP-Routing (localhost)
│   │   │
│   │   └── postgres/
│   │       ├── persistent-volumes.yaml  ← Speicher (bleibt bei Löschen!)
│   │       ├── StatefulSet.yaml         ← Postgres-Container
│   │       ├── Service.yaml             ← Netzwerk-Zugang
│   │       ├── backup-continuous.yaml   ← Auto-Backups (alle 30 Min)
│   │       ├── BACKUP-README.md         ← Backup-Dokumentation
│   │       └── PVC-README.md            ← Persistenz-Dokumentation
│   │
│   └── argocd/
│       └── application.yaml        ← ArgoCD-Konfiguration
│
├── 🔐 SECRETS (LOKAL, NICHT COMMITTEN!)
│   └── secrets/
│       ├── db_password.txt         ← Datenbank-Passwort
│       └── db_user.txt             ← Datenbank-User
│
└── 🤖 AUTOMATION
    ├── pipeline.ps1                ← Build & Push zu Harbor
    ├── backup-now.ps1              ← Manuelles Backup
    ├── backup-to-daemon.ps1        ← Sofort-Backup (persistent)
    └── restore-backup.ps1          ← Backup wiederherstellen
```

🎓 Wichtige Begriffe (Einfach erklärt)

Kubernetes Basics

Begriff	Was ist das?	Analogie
Pod	Ein laufender Container	Wie ein Prozess auf deinem PC
Deployment	Verwaltet mehrere Pods	Wie ein Task-Manager für Pods
Service	Feste Adresse für Pods	Wie eine Domain (immer gleiche Adresse)
Ingress	HTTP-Router von außen	Wie ein Reverse-Proxy (nginx)
Namespace	Ordner in Kubernetes	Wie ein Projekt in Visual Studio
ConfigMap	Nicht-geheime Einstellungen	Wie appsettings.json
Secret	Geheime Daten	Wie Azure Key Vault
PVC	Persistenter Speicher	Wie eine externe Festplatte

GitOps & Tools

Begriff	Was ist das?	Warum wichtig?
GitOps	Git als Single Source of Truth	Alles versioniert & nachvollziehbar
ArgoCD	Automatisches Deployment aus Git	Du pushst Code → ArgoCD deployt
Harbor	Private Docker Registry	Deine Images sicher speichern
kubectl	Kubernetes CLI	Wie `docker` für Kubernetes

🔄 Typischer Workflow (Daily)

Szenario 1: Code geändert

```
# 1. Code ändern (z.B. StudentController.cs)
# ... deine Änderungen ...

# 2. Image neu bauen & pushen
.\pipeline.ps1

# 3. Pods neu starten
kubectl rollout restart deployment/manifest-app -n development

# 4. Warten (~30 Sek)
kubectl rollout status deployment/manifest-app -n development

# 5. Testen
Start http://localhost/swagger
```

Ergebnis: Zero-Downtime Update! (Alte Pods bleiben, bis neue laufen)

Szenario 2: Kubernetes-Config geändert

```
# 1. YAML ändern (z.B. k8s/Student-api/Deployment.yaml)
# ... z.B. replicas: 3 statt 2 ...

# 2. Git Commit & Push
git add .
git commit -m "Scale to 3 replicas"
git push origin cursor

# 3. Warten (~3 Min)
# ArgoCD synct automatisch!

# 4. Prüfen
kubectl get pods -n development
# Sollte jetzt 3x manifest-app zeigen
```

Ergebnis: GitOps in Action! Kein manuelles `kubectl apply` nötig.

🆘 Hilfe & Troubleshooting

Häufige Fehler:

Fehler	Ursache	Lösung
`ImagePullBackOff`	Harbor-Credentials fehlen	Quick-Start Schritt 5
`CrashLoopBackOff`	App startet nicht	`kubectl logs <POD-NAME> -n development`
`OutOfSync` in ArgoCD	Git ≠ Cluster	ArgoCD UI → SYNC
`localhost` funktioniert nicht	Ingress fehlt	Troubleshooting Problem 4

Nützliche Befehle:

```
# Status prüfen
kubectl get all -n development

# Logs ansehen
kubectl logs -n development <POD-NAME>

# Fehler debuggen
kubectl describe pod <POD-NAME> -n development

# Events live sehen
kubectl get events -n development -w

# In Container einsteigen
kubectl exec -it <POD-NAME> -n development -- /bin/sh
```

Vollständiges Troubleshooting:

📖 QUICK-START-GUIDE.md - Troubleshooting

💾 Backups (Datenverlust vermeiden)

Automatische Backups:

✅ **Backup-Daemon läuft** (alle 30 Minuten)
✅ **Letzte 10 Backups** werden behalten
✅ **Backups bleiben** auch bei App-Löschung

```
# Verfügbare Backups ansehen
.\restore-backup.ps1 -ListBackups

# Backup wiederherstellen
.\restore-backup.ps1 -BackupFile continuous-20260203-1200.sql
```

Manuelles Backup:

```
# Sofort-Backup erstellen
.\backup-now.ps1 -BackupName "vor-großer-änderung" -Download

# Backup zum Daemon hochladen
.\backup-to-daemon.ps1 -BackupName "manual-backup"
```

Mehr Details: k8s/postgres/BACKUP-README.md

🔐 Sicherheit (Production Checklist)

⚠️ Für Production wichtig:

```
❌ NIEMALS secrets/ committen (steht in .gitignore)
❌ NIEMALS Passwörter in YAML hardcoden
❌ NIEMALS admin/Harbor12345 in Production nutzen
❌ NIEMALS runAsUser: 0 ohne Grund
❌ NIEMALS imagePullPolicy: Always in Production (besser: Tag mit Version)

✅ IMMER vor Updates: Backup erstellen
✅ IMMER Resource Limits setzen
✅ IMMER Health Checks konfigurieren
✅ IMMER HTTPS nutzen (TLS-Zertifikat)
✅ IMMER RBAC aktivieren
```

Mehr Details: ARCHITECTURE.md - Security Layers

📚 Weiterführende Ressourcen

Offizielle Dokumentation:

- **Kubernetes:** https://kubernetes.io/docs/
- **ArgoCD:** https://argo-cd.readthedocs.io/
- **Harbor:** https://goharbor.io/docs/
- **ASP.NET Core:** https://docs.microsoft.com/aspnet/core/

Tutorials:

- **Kubernetes Basics:** https://kubernetes.io/docs/tutorials/kubernetes-basics/
- **GitOps mit ArgoCD:** https://argo-cd.readthedocs.io/en/stable/getting_started/
- **Docker Best Practices:** https://docs.docker.com/develop/dev-best-practices/

Tools:

- **kubectl Cheat Sheet:** https://kubernetes.io/docs/reference/kubectl/cheatsheet/
- **YAML Validator:** https://www.yamllint.com/
- **JSON zu YAML:** https://www.json2yaml.com/

🎯 Nächste Schritte

Level 1: Anfänger (Du bist hier!)

```
✅ App deployen mit Quick-Start
✅ Grundbegriffe verstehen
✅ Swagger UI nutzen
✅ Backup/Restore testen
```

Level 2: Fortgeschritten

```
□ YAML selbst schreiben/ändern
□ Horizontal Scaling (mehr Replicas)
□ Monitoring mit Prometheus
□ Multi-Environment Setup (dev/staging/prod)
□ CI/CD Pipeline bauen
```

Level 3: Profi

```
□ Helm Charts erstellen
□ Custom Resource Definitions (CRDs)
□ Service Mesh (Istio/Linkerd)
□ GitOps mit Flux CD
□ Kubernetes Operators schreiben
```

✅ Bereit? Los geht's!

Dein nächster Schritt:

👉 **QUICK-START-GUIDE.md** öffnen
👉 **15 Minuten** Zeit nehmen
👉 **Schritt für Schritt** folgen
👉 **http://localhost/swagger** im Browser sehen
👉 **Stolz sein!** 🎉

📞 Support & Feedback

Probleme?

1. **Lies:** QUICK-START-GUIDE - Troubleshooting
2. **Prüfe:** Logs mit `kubectl logs <POD-NAME> -n development`
3. **Checke:** Events mit `kubectl get events -n development`
4. **Suche:** GitHub Issues im Projekt

Fragen?

- **Kubernetes Slack:** https://kubernetes.slack.com/
- **ArgoCD Slack:** https://argoproj.github.io/community/join-slack/
- **Stack Overflow:** Tag `kubernetes`, `argocd`, `harbor`

Viel Erfolg mit deinem Kubernetes-Journey! 🚀

"The journey of a thousand miles begins with a single step."
— Lao Tzu

Dein erster Schritt: QUICK-START-GUIDE.md 👈

==========

DOCUMENTATION-INDEX.md

==========

📚 Dokumentations-Index - Welches Dokument wofür?

🎯 Schnellauswahl

Ich möchte...	Dokument	Dauer	Schwierigkeit
Sofort deployen	QUICK-START-GUIDE.md	15 Min	🟢 Einfach
Alles verstehen	DEPLOYMENT-BLUEPRINT.md	45 Min	🟡 Mittel
YAML lernen	YAML-EXAMPLES.md	60 Min	🟡 Mittel
Architektur sehen	ARCHITECTURE.md	20 Min	🟢 Einfach
Überblick bekommen	GETTING-STARTED.md	10 Min	🟢 Einfach

📖 Dokument-Details

1. GETTING-STARTED.md ⭐ START HIER

Zweck: Einstiegspunkt - Orientierung & Überblick

Inhalt:
- 👋 Willkommen & Einführung
- 📚 Dokumentations-Übersicht
- 📦 Projekt-Übersicht
- 🎓 Wichtige Begriffe
- 🔄 Typischer Workflow
- 🆘 Schnelle Hilfe
- 📚 Weiterführende Ressourcen

Für wen?
- ✅ Absolute Anfänger
- ✅ Erste Orientierung
- ✅ "Was ist das Projekt?"

Empfohlene Reihenfolge: 1. Zuerst

---

2. QUICK-START-GUIDE.md 🚀

Zweck: 15-Minuten-Deployment ohne viel Theorie

Inhalt:
- ✅ Voraussetzungen-Check (5 Min)
- 🚀 10-Schritte-Deployment
- 🔐 Harbor Setup (Klick-für-Klick)
- 🔄 ArgoCD Setup (Klick-für-Klick)
- 🧪 App testen (Swagger)
- 🛠️ Troubleshooting (5 häufigste Fehler)
- 🔁 Daily Workflow
- 🧹 Cleanup
- 💡 Pro-Tipps

Für wen?
- ✅ "Ich will es einfach zum Laufen bringen!"
- ✅ Schnellstart ohne Theorie
- ✅ Copy & Paste Befehle

Empfohlene Reihenfolge: 2. Direkt nach Getting-Started

---

3. DEPLOYMENT-BLUEPRINT.md 📘

Zweck: Vollständige Anleitung mit allen Erklärungen

Inhalt:
- 📦 Projekt-Übersicht (ausführlich)
- 📁 Ordnerstruktur (detailliert)
- 🎓 Begriffe einfach erklärt
- 🎯 Schritt-für-Schritt Deployment (mit Erklärungen)
- 🔐 Harbor Registry Setup (detailliert)
- 🔄 ArgoCD Integration (ausführlich)
- 📝 Workflow (vom Code bis zum Browser)
- 🛠️ Troubleshooting (alle Szenarien)
- 🔧 Nützliche Befehle
- 🆘 Schnelle Hilfe
- 🚨 Sicherheitshinweise

Für wen?
- ✅ "Ich will verstehen, was passiert!"
- ✅ Tieferes Wissen
- ✅ Referenz-Dokument

Empfohlene Reihenfolge: 3. Nach dem Quick-Start

---

4. YAML-EXAMPLES.md 📝

Zweck: Jede YAML-Zeile verstehen lernen

Inhalt:
- 📝 Namespace (vollständig kommentiert)
- 📝 ConfigMap (vollständig kommentiert)
- 📝 Secret (vollständig kommentiert)
- 📝 Deployment (vollständig kommentiert)
- 📝 Service (vollständig kommentiert)
- 📝 Ingress (vollständig kommentiert)
- 📝 PersistentVolumeClaim (vollständig kommentiert)
- 🔗 Wie hängt alles zusammen?
- 💡 Quick Reference (häufige Patterns)

Für wen?
- ✅ "Ich will YAML schreiben lernen!"
- ✅ Kubernetes-Manifeste verstehen
- ✅ Jede Zeile erklärt

Empfohlene Reihenfolge: 4. Parallel zum Blueprint

---

5. ARCHITECTURE.md 🏗️

Zweck: Das große Bild - Wie alles zusammenhängt

Inhalt:
- 📐 System-Architektur Diagramm
- 🔄 Datenfluss (HTTP-Request → DB)
- 🔐 Secrets & ConfigMaps Flow
- 🏗️ Image Build & Deployment Pipeline
- 🔄 GitOps Workflow (visuell)
- 💾 Backup & Recovery Architektur
- 🔒 Security Layers
- 📊 Monitoring & Observability
- 🔧 Technology Stack
- 📈 Skalierungs-Szenarien
- 🌐 Multi-Environment Setup

Für wen?
- ✅ "Ich will das große Bild sehen!"
- ✅ Visuelles Verständnis
- ✅ Architektur-Entscheidungen

Empfohlene Reihenfolge: 5. Optional (zum Vertiefen)

🎯 Lernpfade

Lernpfad 1: Schnellstart (30 Min)

```
1. GETTING-STARTED.md     (10 Min) ← Überblick
2. QUICK-START-GUIDE.md   (15 Min) ← Deployen
3. Swagger testen         (5 Min)  ← Erfolg!
```

Ergebnis: App läuft, grundlegendes Verständnis

Lernpfad 2: Tiefes Verständnis (2-3 Std)

```
1. GETTING-STARTED.md        (10 Min)  ← Überblick
2. QUICK-START-GUIDE.md      (15 Min)  ← Deployen
3. DEPLOYMENT-BLUEPRINT.md   (45 Min)  ← Theorie
4. YAML-EXAMPLES.md          (60 Min)  ← YAML lernen
5. ARCHITECTURE.md           (20 Min)  ← Big Picture
6. Eigene YAMLs schreiben    (30 Min)  ← Praxis
```

Ergebnis: Vollständiges Kubernetes-Verständnis

Lernpfad 3: Troubleshooting (1 Std)

```
1. QUICK-START-GUIDE.md     (15 Min) ← Deployen
2. Absichtlich Fehler bauen (20 Min) ← z.B. falsches Image
3. QUICK-START Troubleshooting (10 Min) ← Fehler fixen
4. kubectl Befehle üben     (15 Min) ← logs, describe, events
```

Ergebnis: Fehler selbstständig debuggen können

📊 Dokument-Vergleich

Dokument	Länge	Level	Praxis	Theorie	Diagramme
GETTING-STARTED	Kurz	🟢	20%	30%	10%
QUICK-START	Mittel	🟢	80%	10%	10%
BLUEPRINT	Lang	🟡	50%	40%	10%
YAML-EXAMPLES	Lang	🟡	30%	70%	20%
ARCHITECTURE	Mittel	🟢	10%	40%	50%

🔍 Suche nach Thema

Thema	Dokument	Kapitel
Harbor Setup	QUICK-START	Schritt 1-3
ImagePullSecret	QUICK-START	Schritt 5
ArgoCD UI	QUICK-START	Schritt 6-7
YAML Syntax	YAML-EXAMPLES	Alle Kapitel
Deployment erklärt	YAML-EXAMPLES	Kapitel 4
Ingress erklärt	YAML-EXAMPLES	Kapitel 6
Datenfluss	ARCHITECTURE	Kapitel 2
Backup	BLUEPRINT	Kapitel Troubleshooting
GitOps	ARCHITECTURE	Kapitel 4
Security	ARCHITECTURE	Kapitel 7
Skalierung	ARCHITECTURE	Kapitel 10
Troubleshooting	QUICK-START	Kapitel Troubleshooting
Begriffe	GETTING-STARTED	Kapitel "Wichtige Begriffe"

🧭 Quick Access Links

Ich habe ein Problem:

Problem	Lösung
App deployed nicht	QUICK-START - Troubleshooting
ImagePullBackOff	QUICK-START - Problem 1
CrashLoopBackOff	QUICK-START - Problem 2
DB verbindet nicht	QUICK-START - Problem 3
localhost nicht erreichbar	QUICK-START - Problem 4
ArgoCD OutOfSync	QUICK-START - Problem 5

Ich will etwas tun:

Aktion	Anleitung
App deployen	QUICK-START - Schritt 1-10
Code ändern & deployen	QUICK-START - Szenario 1
YAML ändern & deployen	QUICK-START - Szenario 2
Backup erstellen	BLUEPRINT - Backups
Backup wiederherstellen	BLUEPRINT - Backups
Skalieren (mehr Pods)	ARCHITECTURE - Scaling

Ich will etwas lernen:

Thema	Ressource
Kubernetes Basics	GETTING-STARTED - Begriffe
YAML schreiben	YAML-EXAMPLES - Alle Kapitel
GitOps verstehen	ARCHITECTURE - GitOps
Architektur verstehen	ARCHITECTURE - Diagramme
Best Practices	BLUEPRINT - Security

🎯 Checkliste: Habe ich alles gelesen?

Minimum (Anfänger):

```
□ GETTING-STARTED.md gelesen
□ QUICK-START-GUIDE durchgeführt
□ App läuft auf localhost/swagger
```

Empfohlen (Fortgeschritten):

```
□ GETTING-STARTED.md gelesen
□ QUICK-START-GUIDE durchgeführt
□ DEPLOYMENT-BLUEPRINT.md gelesen
□ YAML-EXAMPLES teilweise gelesen
□ Eigene YAML-Änderung gemacht
□ Troubleshooting einmal durchgeführt
```

Vollständig (Profi):

```
□ Alle 5 Dokumente gelesen
□ App mehrmals deployed
□ YAMLs selbst geschrieben
□ Backup & Restore getestet
□ Troubleshooting gemeistert
□ Eigene Änderungen gepusht
□ GitOps-Workflow verstanden
```

📞 Noch Fragen?

Weitere Ressourcen:

- **Projekt-spezifisch:**
- k8s/postgres/BACKUP-README.md - Backup-System
- k8s/postgres/PVC-README.md - Persistenz
- **Offizielle Docs:**
- Kubernetes Docs
- ArgoCD Docs
- Harbor Docs
- **Tutorials:**
- Kubernetes Basics
- GitOps Guide

Bereit? Starte mit: GETTING-STARTED.md 👈

Zuletzt aktualisiert: 2026-02-03

==========

QUICK-START-GUIDE.md

==========

⚡ Quick Start Guide - Von 0 auf 100 in 15 Minuten

🎯 Ziel

Deine Student-API läuft und ist erreichbar unter `http://localhost/swagger`

✅ Voraussetzungen (5 Min)

1. Docker Desktop läuft?

```
docker version
```

✅ Sollte Version-Infos zeigen (Client + Server)
❌ Fehler? → Docker Desktop starten

2. Kubernetes aktiv?

```
kubectl get nodes
```

✅ Sollte `docker-desktop Ready` zeigen
❌ Fehler? → Docker Desktop → Settings → Kubernetes → ✓ Enable Kubernetes

3. Harbor läuft?

Browser öffnen: `http://localhost:30002`
✅ Login-Seite sichtbar
❌ Fehler? → Harbor neu installieren

4. ArgoCD installiert?

```
kubectl get pods -n argocd
```

✅ Zeigt mehrere Pods (alle Running)
❌ Fehler? → ArgoCD installieren:

```
kubectl create namespace argocd
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/inst
```

🚀 Deployment in 10 Schritten

Schritt 1: Harbor-Projekt erstellen (1 Min)

Was: Erstelle einen Ordner für deine Images in Harbor

Wo: Browser → `http://localhost:30002`

Wie:
1. Login:
- Username: `admin`
- Password: `Harbor12345`
2. Klicke: **"+ NEW PROJECT"**
3. Eingeben:
- Project Name: `studenten`
- Access Level: **Public** (oder Private mit Registrierung)
4. Klicke: **"OK"**

✅ **Erfolgreich wenn:** Du siehst "studenten" in der Projektliste

---

Schritt 2: Docker bei Harbor anmelden (30 Sek)

Was: Docker authentifizieren, damit `push` funktioniert

PowerShell öffnen:

```
docker login localhost:30002
```

Eingeben:

```
Username: admin
Password: Harbor12345
```

✅ **Erfolgreich wenn:** `Login Succeeded`

Schritt 3: Image bauen & pushen (2 Min)

Was: Deine App als Docker-Image verpacken und zu Harbor hochladen

```
# Zum Projektordner
cd C:\Users\hedin\source\repos\manifest\manifest\manifest

# Automatisches Build & Push
.\pipeline.ps1
```

Was passiert im Hintergrund:

```
# 1. Image bauen
docker build -t localhost:30002/studenten/manifest-app:latest ./StudentApi

# 2. Mit Zeitstempel taggen
docker tag localhost:30002/studenten/manifest-app:latest \
  localhost:30002/studenten/manifest-app:v1-20260203-1154

# 3. Zu Harbor pushen
docker push localhost:30002/studenten/manifest-app:latest
docker push localhost:30002/studenten/manifest-app:v1-20260203-1154
```

✅ **Erfolgreich wenn:**
`Done! Pushed localhost:30002/studenten/manifest-app:v1-...`

Prüfen in Harbor:
1. Browser: `http://localhost:30002`
2. Klicke: **Projects** → `studenten` → **Repositories**
3. Du solltest sehen: `manifest-app` mit Tag `latest`

Schritt 4️⃣: Kubernetes-Namespace erstellen (10 Sek)

Was: Erstelle den "Ordner" in Kubernetes

```
kubectl create namespace development
```

✅ **Erfolgreich wenn:** `namespace/development created`

Oder falls schon existiert:

```
Error from server (AlreadyExists): namespaces "development" already exists
```

→ Das ist OK! ✅

Schritt 5️⃣: ImagePullSecret erstellen (30 Sek)

Was: Passwort für Kubernetes, damit es dein Image von Harbor holen kann

```
kubectl create secret docker-registry harbor-regcred `
  --docker-server=localhost:30002 `
  --docker-username=admin `
  --docker-password=Harbor12345 `
  --docker-email=admin@local `
  --namespace=development
```

Befehl erklärt:
- `docker-registry` : Typ des Secrets (für Container-Registries)
- `harbor-regcred` : Name (muss in `Deployment.yaml` unter `imagePullSecrets` stehen!)
- `--docker-server` : Adresse deiner Harbor-Instanz
- `--docker-username` : Dein Harbor-Login
- `--docker-password` : Dein Harbor-Passwort
- `--namespace` : In welchem Namespace

✅ **Erfolgreich wenn:** `secret/harbor-regcred created`

Prüfen:

```
kubectl get secret harbor-regcred -n development
```

Sollte zeigen: `harbor-regcred kubernetes.io/dockerconfigjson 1 5s`

Schritt 6️⃣: ArgoCD UI öffnen (1 Min)

Was: Zugriff auf ArgoCD Web-Interface

A) Passwort abrufen

```
$secret = kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}"
[System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String($secret))
```

Kopiere das Passwort! (z.B. `xY3k9mP4qR7s`)

B) Port-Forward starten

```
kubectl port-forward svc/argocd-server -n argocd 8080:443
```

Lass das Terminal offen! (Port-Forward läuft bis du STRG+C drückst)

C) Browser öffnen

```
https://localhost:8080
```

Warnung ignorieren:
"Your connection is not private" → **Erweitert** → **Fortfahren zu localhost**

Login:
- Username: `admin`
- Password: (das von oben)

✅ **Erfolgreich wenn:** Du siehst das ArgoCD Dashboard

Schritt 7️⃣: Application in ArgoCD erstellen (2 Min)

Was: Sag ArgoCD, dass es dein Git-Repo überwachen soll

Option A: Via UI (empfohlen für Anfänger)

Oben links: Klicke **"+ NEW APP"**

Formular ausfüllen:

📍 **GENERAL**

```
Application Name:  manifest-app
Project Name:      default          [Dropdown]
Sync Policy:       [Toggle] Automatic
  ✓ PRUNE RESOURCES
  ✓ SELF HEAL
```

📍 **SOURCE**

```
Repository URL:    https://github.com/Muhi94/manifest.git
Revision:          cursor           (oder HEAD für main)
Path:              k8s
```

📍 **DESTINATION**

```
Cluster URL:       https://kubernetes.default.svc   [Dropdown]
Namespace:         development
```

📍 **SYNC OPTIONS (erweitern)**

```
✓ AUTO-CREATE NAMESPACE
✓ SERVER SIDE APPLY
```

Unten: Klicke **"CREATE"**

Option B: Via Terminal (schneller)

```
kubectl apply -f argocd/application.yaml
```

✅ **Erfolgreich wenn:** `application.argoproj.io/manifest-app created`

Schritt 8️⃣: Synchronisation prüfen (2 Min)

In ArgoCD UI:

Du siehst jetzt eine Karte: **manifest-app**

Status-Entwicklung:

```
1. OutOfSync → Syncing → Synced      ✅
2. Missing   → Progressing → Healthy ✅
```

Dauer: 1-3 Minuten

Was passiert gerade?
- Namespace `development` wird erstellt
- Postgres-Datenbank wird gestartet
- 2x App-Pods werden gestartet
- Backup-Daemon wird gestartet
- Ingress wird konfiguriert

Live-Ansicht:

Klicke auf die **manifest-app** Karte → Du siehst eine Grafik mit allen Ressourcen:

```
manifest-app (Application)
 ├─ Namespace: development
 ├─ ConfigMap: app-config
 ├─ Secret: db-credentials
 ├─ Deployment: manifest-app (2 Pods)
 ├─ Service: manifest-app-service
 ├─ Ingress: manifest-app-ingress
 ├─ StatefulSet: postgres (1 Pod)
 ├─ Service: postgres-service
 ├─ PVC: postgres-data-postgres-0
 ├─ Deployment: postgres-backup-daemon
 └─ PVC: postgres-backup-pvc
```

Schritt 9️⃣: Pods prüfen (1 Min)

Terminal öffnen:

```
kubectl get pods -n development
```

Sollte zeigen (alle READY 1/1, STATUS Running):

```
NAME                                    READY   STATUS    RESTARTS   AGE
manifest-app-7f75894c77-abc12           1/1     Running   0          2m
manifest-app-7f75894c77-def34           1/1     Running   0          2m
postgres-0                              1/1     Running   0          2m
postgres-backup-daemon-569446944d-ghi56 1/1     Running   0          2m
```

✅ **Alle Running?** → Weiter zu Schritt 10!

❌ **Fehler? (z.B. ImagePullBackOff, CrashLoopBackOff)**

```
# Details ansehen
kubectl describe pod <POD-NAME> -n development

# Logs ansehen
kubectl logs <POD-NAME> -n development
```

Häufige Fehler → Siehe Troubleshooting unten

Schritt 🔟: App testen! (2 Min)

🌐 **Browser öffnen:**

```
http://localhost/swagger
```

✅ **Erfolgreich wenn:** Swagger UI wird geladen mit Endpunkten:
- `GET /api/student`
- `POST /api/student`
- `DELETE /api/student/{id}`

📝 **Student anlegen (POST):**

1. Klicke auf: **POST** `/api/student`
2. Klicke: **"Try it out"**
3. Ändere JSON:
   `json { "name": "Max Mustermann", "age": 25 }`
4. Klicke: **"Execute"**
5. Response Code: **201 Created** ✅

📖 **Student abrufen (GET):**

1. Klicke auf: **GET** `/api/student`
2. Klicke: **"Try it out"**
3. Klicke: **"Execute"**
4. Response:
   `json [ { "id": 1, "name": "Max Mustermann", "age": 25 } ]`

🎉 GRATULATION!

Deine App läuft in Kubernetes! 🚀

📊 Status-Übersicht

Alles auf einen Blick:

```
kubectl get all -n development
```

Sollte zeigen:

```
NAME                                          READY   STATUS
pod/manifest-app-7f75894c77-xxxxx             1/1     Running
pod/manifest-app-7f75894c77-yyyyy             1/1     Running
pod/postgres-0                                1/1     Running
pod/postgres-backup-daemon-569446944d-zzzzz   1/1     Running

NAME                           TYPE        CLUSTER-IP      PORT(S)
service/manifest-app-service   ClusterIP   10.96.x.x       80/TCP
service/postgres-service       ClusterIP   10.96.x.x       5432/TCP
service/postgres-headless      ClusterIP   None            5432/TCP

NAME                                     READY   UP-TO-DATE   AVAILABLE
deployment.apps/manifest-app             2/2     2            2
deployment.apps/postgres-backup-daemon   1/1     1            1

NAME                                                DESIRED   CURRENT   READY
replicaset.apps/manifest-app-7f75894c77             2         2         2
replicaset.apps/postgres-backup-daemon-569446944d   1         1         1

NAME                         READY
statefulset.apps/postgres    1/1
```

🔄 Daily Workflow: Änderungen deployen

Szenario 1: Code geändert (App-Logik)

```
# 1. Code ändern in StudentApi/
# ... deine Änderungen ...

# 2. Neu bauen & pushen
.\pipeline.ps1

# 3. Pods neu starten
kubectl rollout restart deployment/manifest-app -n development

# 4. Warten
kubectl rollout status deployment/manifest-app -n development

# 5. Testen
http://localhost/swagger
```

Szenario 2: Kubernetes-Config geändert (z.B. Replicas)

```
# k8s/Student-api/Deployment.yaml
spec:
  replicas: 3  # War vorher 2
```

```
# 1. Git Commit & Push
git add k8s/Student-api/Deployment.yaml
git commit -m "Scale to 3 replicas"
git push origin cursor
```

```
# 2. Warten (ArgoCD synct automatisch in ~3 Min)
# ODER: In ArgoCD UI auf "SYNC" klicken

# 3. Prüfen
kubectl get pods -n development
# Sollte jetzt 3x manifest-app Pods zeigen
```

🛠️ Troubleshooting

❌ Problem 1: ImagePullBackOff

Symptom:

```
manifest-app-xxx   0/1   ImagePullBackOff
```

Ursache: Kubernetes kann Image nicht von Harbor holen

Lösung:

```
# 1. Prüfe Secret
kubectl get secret harbor-regcred -n development

# Falls nicht existiert:
kubectl create secret docker-registry harbor-regcred \
  --docker-server=localhost:30002 \
  --docker-username=admin \
  --docker-password=Harbor12345 \
  --docker-email=admin@local \
  -n development

# 2. Pod löschen (wird neu erstellt)
kubectl delete pod -n development -l app=manifest-app

# 3. Prüfen
kubectl get pods -n development -w
```

❌ Problem 2: CrashLoopBackOff (Postgres)

Symptom:

```
postgres-0   0/1   CrashLoopBackOff
```

Ursache: Postgres kann nicht auf Daten-Volume schreiben

Lösung:

```
# Logs ansehen
kubectl logs postgres-0 -n development

# Häufig: Permission-Fehler
# → StatefulSet nutzt runAsUser: 0 (Root) für hostpath

# PVC neu erstellen
kubectl delete pvc postgres-data-postgres-0 -n development
kubectl delete pod postgres-0 -n development
```

❌ Problem 3: App verbindet nicht zur DB

Symptom in Logs:

```
Database not ready yet, retrying...
```

Lösung:

```
# 1. Prüfe ob Postgres läuft
kubectl get pods -n development postgres-0

# 2. Teste Verbindung manuell
kubectl run pg-test --rm -i --restart=Never -n development \
  --image=postgres:15-alpine \
  --env=PGPASSWORD=SuperSecurePassword123! \
  -- psql -h postgres-service -U app_user -d studentdb -c 'SELECT 1;'

# Sollte zeigen:
#  ?column?
# ----------
#         1

# 3. Falls Fehler: Secret prüfen
kubectl get secret db-credentials -n development -o yaml
```

❌ Problem 4: localhost funktioniert nicht

Symptom:

```
Diese Seite kann nicht aufgerufen werden
```

Ursache: Ingress Controller fehlt

Lösung:

```
# 1. Prüfe Ingress
kubectl get ingress -n development

# 2. Prüfe Ingress Controller
kubectl get pods -n ingress-nginx

# Falls keine Pods:
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/main/deploy/static/prov

# Warte bis Running:
kubectl wait --namespace ingress-nginx \
  --for=condition=ready pod \
  --selector=app.kubernetes.io/component=controller \
  --timeout=120s

# Alternative: Port-Forward direkt zur App
kubectl port-forward svc/manifest-app-service -n development 5000:80
# Dann: http://localhost:5000/swagger
```

❌ Problem 5: ArgoCD zeigt OutOfSync

Ursache: Manuelle Änderung in Kubernetes oder Git nicht aktuell

Lösung:

```
# In ArgoCD UI:
# 1. Klicke auf manifest-app
# 2. Klicke "SYNC"
# 3. Wähle "SYNCHRONIZE"

# ODER im Terminal:
kubectl patch app manifest-app -n argocd \
  --type merge \
  -p '{"metadata":{"annotations":{"argocd.argoproj.io/refresh":"hard"}}}'
```

🧹 Cleanup: Alles löschen

Option 1: Nur die App (Daten bleiben!)

```
# Application löschen
kubectl delete application manifest-app -n argocd

# PVCs bleiben bestehen (Daten safe!)
kubectl get pvc -n development
```

Option 2: Alles inkl. Daten

```
# ⚠️ WARNUNG: ALLE DATEN GEHEN VERLOREN!
# Erst Backup machen:
.\backup-now.ps1 -BackupName "before-delete" -Download

# Dann löschen:
kubectl delete namespace development
kubectl delete application manifest-app -n argocd
```

Option 3: Nur Pods neu starten (Daten bleiben)

```
# Nur App neu starten
kubectl rollout restart deployment/manifest-app -n development

# Alles neu starten
kubectl delete pod --all -n development
```

📚 Weiterführende Guides

- **Vollständige Dokumentation:** `DEPLOYMENT-BLUEPRINT.md`
- **YAML-Erklärungen:** `YAML-EXAMPLES.md`
- **Backup-System:** `k8s/postgres/BACKUP-README.md`
- **Persistenz:** `k8s/postgres/PVC-README.md`

🎯 Checkliste: Ist alles ready?

```
□ Harbor erreichbar (localhost:30002)
□ Image sichtbar in Harbor (studenten/manifest-app:latest)
```

```
□ Namespace existiert (kubectl get ns development)
□ Secret existiert (kubectl get secret harbor-regcred -n development)
□ ArgoCD UI erreichbar (localhost:8080)
□ Application in ArgoCD: Status "Synced", Health "Healthy"
□ Alle Pods Running (kubectl get pods -n development)
□ Swagger öffnet (localhost/swagger)
□ POST /api/student funktioniert
□ GET /api/student zeigt Daten
□ Backup-Daemon läuft (kubectl logs -n development -l app=postgres-backup)
```

Alle ✓? → Du bist fertig! 🎉

💡 Pro-Tipps

Tip 1: Watch-Mode für Live-Updates

```
# Pods live beobachten
kubectl get pods -n development -w

# Events live sehen
kubectl get events -n development -w

# Logs live folgen
kubectl logs -f -n development -l app=manifest-app
```

Tip 2: Schneller Port-Forward

```
# App direkt erreichen (ohne Ingress)
kubectl port-forward svc/manifest-app-service -n development 5000:80
# → http://localhost:5000/swagger

# Postgres direkt erreichen (z.B. mit pgAdmin)
kubectl port-forward svc/postgres-service -n development 5432:5432
# → Host: localhost, Port: 5432, User: app_user
```

Tip 3: Alias für häufige Befehle

```
# In PowerShell-Profil ($PROFILE) eintragen:
function k { kubectl $args }
function kgp { kubectl get pods -n development $args }
function kl { kubectl logs -n development $args }
function kd { kubectl describe -n development $args }

# Dann nutzbar:
k get pods -n development  # → kgp
k logs pod-name            # → kl pod-name
```

Tip 4: Backup vor jedem großen Update

```
# Automatisch mit Zeitstempel
.\backup-now.ps1 -Download
```

```
# Oder als Datum
.\backup-now.ps1 -BackupName "vor-update-$(Get-Date -Format 'yyyyMMdd')" -Download
```

Viel Erfolg! 🚀

==========

DEPLOYMENT-BLUEPRINT.md

==========

🚀 Kubernetes Deployment Blueprint für Anfänger

Student API - Vollständige Deployment-Anleitung

📚 Inhaltsverzeichnis

📦 Projekt-Übersicht

Was ist das?
Eine Student-API (ASP.NET Core) mit Postgres-Datenbank, die in Kubernetes läuft.

Was macht die App?
- Verwaltet Studenten-Datensätze (Erstellen, Lesen, Löschen)
- Bietet eine REST API auf Port 8080
- Hat eine Swagger-UI zum Testen: `http://localhost/swagger`

Was brauchst du?
- Docker Desktop mit Kubernetes aktiviert
- Harbor Registry (läuft auf `localhost:30002`)
- ArgoCD installiert im Cluster
- Git Repository (dein aktuelles Projekt)

📁 Ordnerstruktur

```
manifest/
├── StudentApi/                  # Deine .NET Anwendung
│   ├── Dockerfile               # Bauplan für das Docker-Image
│   ├── Program.cs               # Hauptcode der App
│   └── ...
│
├── k8s/                         # Alle Kubernetes-Dateien
│   ├── Student-api/             # Deine API-Konfiguration
│   │   ├── Namespace.yaml       # Erstellt den "Arbeitsbereich" in Kubernetes
```

```
│   │   ├── ConfigMap.yaml        # Nicht-geheime Einstellungen (DB-Host, DB-Name)
│   │   ├── Deployment.yaml       # Wie deine App läuft (2 Kopien, Ressourcen, etc.)
│   │   └── Ingress.yaml          # Macht die App von außen erreichbar (localhost)
│   │
│   └── postgres/                 # Datenbank-Konfiguration
│       ├── persistent-volumes.yaml    # Speicher für Datenbank (bleibt bei Löschen!)
│       ├── StatefulSet.yaml           # Postgres-Container
│       ├── Service.yaml               # Netzwerk-Zugang zur Datenbank
│       ├── backup-continuous.yaml     # Automatische Backups alle 30 Min
│       ├── BACKUP-README.md           # Backup-Dokumentation
│       └── PVC-README.md              # Persistenz-Dokumentation
│
├── argocd/                       # ArgoCD-Konfiguration
│   └── application.yaml          # Sagt ArgoCD: "Deploy alles aus k8s/"
│
├── secrets/                      # Passwörter (NICHT committen!)
│   ├── db_password.txt
│   └── db_user.txt
│
├── pipeline.ps1                  # Script zum Image bauen & pushen
├── backup-now.ps1                # Manuelles Backup
├── backup-to-daemon.ps1          # Sofort-Backup
└── restore-backup.ps1            # Backup wiederherstellen
```

🎓 Wichtige Begriffe einfach erklärt

Kubernetes Basics

Begriff	Was ist das?	Beispiel aus deinem Projekt
Pod	Ein Container, der läuft	`manifest-app-7f75894c77-99d6g` (deine App läuft drin)
Namespace	Ein "Ordner" in Kubernetes zur Trennung	`development` (alle deine Ressourcen sind darin)
Deployment	Sagt: "Starte 2 Kopien meiner App"	`k8s/Student-api/Deployment.yaml` (2 replicas)
Service	Eine feste Adresse, um Pods zu erreichen	`manifest-app-service` (zeigt auf deine App-Pods)
Ingress	Der "Türsteher" - macht Apps von außen erreichbar	`localhost` → zu deiner App
PVC	Speicher, der bleibt (wie eine externe Festplatte)	`postgres-data-postgres-0` (10GB für DB-Daten)
ConfigMap	Nicht-geheime Einstellungen	DB-Host: `postgres-service`
Secret	Geheime Daten (Passwörter)	DB-User: `app_user`, Passwort: `SuperSecure...`

GitOps & Tools

Begriff	Was ist das?	Warum brauchst du es?
GitOps	Kubernetes liest Konfiguration aus Git	Du änderst Code → Git Push → Kubernetes deployt automatisch
ArgoCD	Ein Tool, das GitOps umsetzt	Überwacht dein Git-Repo und synchronisiert Kubernetes
Harbor	Private Docker Registry (wie Docker Hub, nur lokal)	Speichert deine selbst gebauten Docker-Images
ImagePullSecret	Passwort für Harbor	Damit Kubernetes dein Image herunterladen darf

---

🎯 Schritt-für-Schritt Deployment

Phase 1: Vorbereitung (einmalig)

Schritt 1: Prüfe ob alles läuft

```
# Docker Desktop läuft?
docker version

# Kubernetes aktiv?
kubectl get nodes
# Sollte zeigen: docker-desktop   Ready

# Harbor läuft?
# Öffne Browser: http://localhost:30002
# Login: admin / Harbor12345
```

Schritt 2: Harbor - Projekt erstellen

1. Öffne: `http://localhost:30002`
2. Login: `admin` / `Harbor12345`
3. Klicke: **"New Project"**
4. Name: `studenten`
5. Access Level: **Public** (für einfaches Testing) oder **Private**
6. Klicke: **"OK"**

Phase 2: Image bauen & pushen

Schritt 3: Bei Harbor anmelden

```
# Terminal öffnen
docker login localhost:30002

# Eingeben:
# Username: admin
# Password: Harbor12345
```

Was passiert?
Docker speichert deine Anmeldedaten, damit `docker push` funktioniert.

Schritt 4: Image bauen & hochladen

```
# Zum Projektordner wechseln
cd C:\Users\hedin\source\repos\manifest\manifest\manifest

# Automatisches Build & Push
.\pipeline.ps1
```

Was macht `pipeline.ps1` ?
1. Baut deine App als Docker-Image
2. Taggt es mit Zeitstempel (z.B. `v1-20260203-1154`)
3. Pusht zu Harbor: `localhost:30002/studenten/manifest-app:latest`

Erfolgreich wenn du siehst:

```
Done! Pushed localhost:30002/studenten/manifest-app:v1-XXXXXXXX-XXXX
```

Schritt 5: Prüfe in Harbor

1. Browser: `http://localhost:30002`
2. Gehe zu: **Projects** → `studenten`
3. Klicke: **Repositories**
4. Du solltest sehen: `manifest-app` mit Tag `latest`

Phase 3: Kubernetes Secrets erstellen

Schritt 6: ImagePullSecret erstellen

Was ist das?
Ein Passwort, damit Kubernetes dein Image von Harbor herunterladen darf.

```
kubectl create secret docker-registry harbor-regcred \
  --docker-server=localhost:30002 \
  --docker-username=admin \
  --docker-password=Harbor12345 \
  --docker-email=admin@local \
  -n development
```

Wenn Namespace noch nicht existiert:

```
kubectl create namespace development
```

Prüfen:

```
kubectl get secret harbor-regcred -n development
# Sollte zeigen: harbor-regcred   kubernetes.io/dockerconfigjson
```

Phase 4: ArgoCD Setup

Schritt 7: ArgoCD UI öffnen

```
# Finde ArgoCD Passwort
kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" |
  ForEach-Object { [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String($_)) }
```

```
# Port-Forward zu ArgoCD
kubectl port-forward svc/argocd-server -n argocd 8080:443
```

Öffne Browser:
- URL: `https://localhost:8080`
- Username: `admin`
- Password: (das aus dem Befehl oben)
- **Warnung ignorieren:** "Your connection is not private" → Fortfahren

Schritt 8: Application in ArgoCD erstellen

Option A: Via UI (einfacher für Anfänger)

1. Klicke oben links: **"+ NEW APP"**
2. **GENERAL**
3. Application Name: `manifest-app`
4. Project: `default`
5. Sync Policy: `Automatic` ✓
   - ✓ `PRUNE RESOURCES`
   - ✓ `SELF HEAL`
6. **SOURCE**
7. Repository URL: `https://github.com/Muhi94/manifest.git`
8. Revision: `cursor` (oder `HEAD` für main-Branch)
9. Path: `k8s`
10. **DESTINATION**
11. Cluster URL: `https://kubernetes.default.svc` (sollte schon drin sein)
12. Namespace: `development`
13. **SYNC POLICY** (erweitert)
14. Sync Options: ✓ `AUTO-CREATE NAMESPACE`
15. Klicke: **"CREATE"**

Option B: Via Terminal (schneller)

```
kubectl apply -f argocd/application.yaml
```

Schritt 9: Warte auf Sync

In der ArgoCD UI siehst du jetzt:
- **Status:** `OutOfSync` → `Syncing` → `Synced`
- **Health:** `Progressing` → `Healthy`

Dauer: 1-3 Minuten

Was passiert gerade?
- ArgoCD liest dein Git-Repo
- Erstellt Namespace `development`
- Deployt Postgres (Datenbank)
- Deployt deine App (2 Pods)

- Startet Backup-Daemon
- Erstellt Ingress (localhost)

Phase 5: Prüfen ob alles läuft

Schritt 10: Pods prüfen

```
kubectl get pods -n development
```

Sollte zeigen (alle READY 1/1):

```
NAME                                  READY   STATUS
manifest-app-7f75894c77-xxxxx         1/1     Running
manifest-app-7f75894c77-yyyyy         1/1     Running
postgres-0                            1/1     Running
postgres-backup-daemon-569446944d-zzzzz   1/1     Running
```

Falls STATUS nicht `Running`:

```
# Zeige Details
kubectl describe pod <POD-NAME> -n development

# Zeige Logs
kubectl logs <POD-NAME> -n development
```

Schritt 11: Services prüfen

```
kubectl get svc -n development
```

Sollte zeigen:

```
NAME                  TYPE        CLUSTER-IP      PORT(S)
manifest-app-service  ClusterIP   10.x.x.x        80
postgres-service      ClusterIP   10.x.x.x        5432
postgres-headless     ClusterIP   None            5432
```

Schritt 12: Ingress prüfen

```
kubectl get ingress -n development
```

Sollte zeigen:

```
NAME                  CLASS   HOSTS       ADDRESS     PORTS
manifest-app-ingress  nginx   localhost   localhost   80
```

Phase 6: App testen

Schritt 13: Swagger UI öffnen

Browser öffnen:

```
http://localhost/swagger
```

Du solltest sehen:
- Swagger UI mit `/api/Student` Endpoints
- GET, POST, DELETE Operationen

Schritt 14: Ersten Student anlegen

In Swagger UI:

1. Klicke auf: **POST** `/api/student`
2. Klicke: **"Try it out"**
3. Ändere JSON:
   `json { "name": "Max Mustermann", "age": 25 }`
4. Klicke: **"Execute"**
5. **Response:** `201 Created` ✓

Schritt 15: Studenten abrufen

1. Klicke auf: **GET** `/api/student`
2. Klicke: **"Try it out"**
3. Klicke: **"Execute"**
4. **Response:**
   `json [ { "id": 1, "name": "Max Mustermann", "age": 25 } ]`

🎉 **Gratulation! Deine App läuft!**

🔐 Harbor Registry Setup (Detailliert)

Was ist Harbor?

Ein privater Ort zum Speichern deiner Docker-Images (wie eine private Cloud für Container).

Warum brauchst du das?

- Docker Hub hat Rate Limits (max. 100 Pulls/6h)
- Firmen-Images sollten nicht öffentlich sein
- Du hast volle Kontrolle

ImagePullSecret erstellen

Was ist das?
Ein Passwort-Tresor, den Kubernetes nutzt, um dein Image von Harbor herunterzuladen.

Befehl:

```
kubectl create secret docker-registry harbor-regcred \
  --docker-server=localhost:30002 \
  --docker-username=admin \
  --docker-password=Harbor12345 \
  --docker-email=admin@local \
  --namespace=development
```

Parameter erklärt:
- `docker-registry` : Typ des Secrets (für Docker Registries)
- `harbor-regcred` : Name des Secrets (frei wählbar, muss in Deployment.yaml passen)
- `--docker-server` : Adresse deiner Harbor-Instanz
- `--docker-username` : Dein Harbor-Login
- `--docker-password` : Dein Harbor-Passwort

- `--docker-email`: Beliebig (wird nicht geprüft)
- `--namespace`: In welchem Namespace das Secret erstellt wird

Prüfen ob es geklappt hat:

```
kubectl get secret harbor-regcred -n development -o yaml
```

Sollte zeigen:

```
type: kubernetes.io/dockerconfigjson
data:
  .dockerconfigjson: eyJ... (verschlüsselt)
```

🔄 ArgoCD Integration (Klick-für-Klick)

Was ist ArgoCD?

Ein Tool, das dein Git-Repository überwacht. Wenn du Code änderst und pushst, deployt ArgoCD automatisch die neuen Versionen in Kubernetes.

Wie funktioniert das?

1. Du änderst eine Datei in `k8s/` (z.B. `Deployment.yaml`)
2. `git add` → `git commit` → `git push`
3. ArgoCD sieht die Änderung (alle 3 Minuten)
4. ArgoCD wendet die neue Konfiguration an
5. Kubernetes startet neue Pods mit der neuen Version

Application anlegen (UI-Anleitung)

1. ArgoCD UI öffnen

```
# Falls noch nicht geöffnet:
kubectl port-forward svc/argocd-server -n argocd 8080:443
```

Browser: `https://localhost:8080`
Login: `admin` / (Passwort von oben)

2. Neue Application erstellen

Oben links: Klicke **"+ NEW APP"**

3. Formular ausfüllen

GENERAL Section:

```
Application Name: manifest-app
Project Name:     default (aus Dropdown)
Sync Policy:      Automatic (Toggle aktivieren)
```

Wenn Automatic aktiviert:
- ✓ Haken bei: `PRUNE RESOURCES`
(Bedeutet: Lösche Ressourcen, die nicht mehr im Git sind)
- ✓ Haken bei: `SELF HEAL`
(Bedeutet: Repariere automatisch, wenn jemand manuell was ändert)

SOURCE Section:

```
Repository URL:    https://github.com/Muhi94/manifest.git
Revision:          cursor        (dein Branch-Name)
Path:              k8s           (Ordner mit allen YAML-Dateien)
```

DESTINATION Section:

```
Cluster URL:       https://kubernetes.default.svc   (aus Dropdown)
Namespace:         development
```

SYNC OPTIONS (erweitern):
- ✓ Haken bei: `AUTO-CREATE NAMESPACE`

4. Erstellen

Unten: Klicke **"CREATE"**

5. Synchronisation starten

Falls App nicht automatisch synct:
- Klicke auf die App-Karte
- Oben: Klicke **"SYNC"**
- Im Dialog: Klicke **"SYNCHRONIZE"**

📝 Workflow - Vom Code bis zum Browser

Komplette Checkliste

```
□ Schritt 1:  Harbor öffnen (localhost:30002) → Projekt "studenten" existiert?
□ Schritt 2:  docker login localhost:30002 (admin / Harbor12345)
□ Schritt 3:  .\pipeline.ps1 (baut & pusht Image)
□ Schritt 4:  kubectl create namespace development (falls nicht existiert)
□ Schritt 5:  kubectl create secret docker-registry harbor-regcred ... (siehe oben)
□ Schritt 6:  kubectl get secret harbor-regcred -n development (prüfen)
□ Schritt 7:  ArgoCD UI öffnen (localhost:8080)
□ Schritt 8:  Application erstellen (Formular ausfüllen, siehe oben)
□ Schritt 9:  Warten auf SYNC (1-3 Min)
□ Schritt 10: kubectl get pods -n development (alle Running?)
□ Schritt 11: Browser: http://localhost/swagger
□ Schritt 12: POST /api/student → Student anlegen
□ Schritt 13: GET /api/student → Student sehen
□ Schritt 14: 🎉 Fertig!
```

🔄 Änderungen deployen (Daily Workflow)

Szenario: Du änderst Code in der App

```
# 1. Code ändern (z.B. StudentController.cs)
# ... deine Änderungen ...

# 2. Image neu bauen
.\pipeline.ps1

# 3. Pods neu starten (damit sie das neue Image ziehen)
```

```
kubectl rollout restart deployment/manifest-app -n development

# 4. Warten
kubectl rollout status deployment/manifest-app -n development

# 5. Testen
# Browser: http://localhost/swagger
```

Szenario: Du änderst Kubernetes-Config

```
# 1. YAML ändern (z.B. k8s/Student-api/Deployment.yaml)
# ... deine Änderungen ...

# 2. Git Commit & Push
git add .
git commit -m "Update deployment replicas to 3"
git push origin cursor

# 3. Warten (ArgoCD synct automatisch in 3 Min)
# ODER: In ArgoCD UI auf "SYNC" klicken

# 4. Prüfen
kubectl get pods -n development
```

🛠️ Troubleshooting

Problem: Pods starten nicht (ImagePullBackOff)

Symptom:

```
manifest-app-xxx   0/1   ImagePullBackOff
```

Ursache: Kubernetes kann Image nicht von Harbor holen

Lösung:

```
# 1. Prüfe ob harbor-regcred existiert
kubectl get secret harbor-regcred -n development

# Falls nicht:
kubectl create secret docker-registry harbor-regcred \
  --docker-server=localhost:30002 \
  --docker-username=admin \
  --docker-password=Harbor12345 \
  --docker-email=admin@local \
  -n development

# 2. Prüfe ob Image in Harbor existiert
# Browser: localhost:30002 → Projects → studenten → Repositories

# 3. Pod neu starten
kubectl delete pod -n development -l app=manifest-app
```

Problem: Pods CrashLoopBackOff

Symptom:

```
manifest-app-xxx   0/1   CrashLoopBackOff
```

Ursache: App startet, aber stürzt sofort ab

Lösung:

```
# Logs ansehen
kubectl logs -n development <POD-NAME>

# Häufige Ursachen:
# - Datenbank nicht erreichbar
# - Falsches Secret
# - Port schon belegt
```

Problem: Datenbank verbindet nicht

Symptom in Logs:

```
Database not ready yet, retrying...
```

Lösung:

```
# 1. Prüfe ob Postgres läuft
kubectl get pods -n development postgres-0
# Sollte: Running (1/1)

# 2. Prüfe Postgres-Logs
kubectl logs -n development postgres-0

# 3. Prüfe Secret
kubectl get secret db-credentials -n development -o yaml

# 4. Teste Verbindung manuell
kubectl run pg-test --rm -i --restart=Never -n development \
  --image=postgres:15-alpine \
  --env=PGPASSWORD=SuperSecurePassword123! \
  -- psql -h postgres-service -U app_user -d studentdb -c 'SELECT 1;'
```

Problem: localhost funktioniert nicht im Browser

Symptom:

```
Diese Seite kann nicht aufgerufen werden
```

Ursache: Ingress Controller fehlt oder falsch konfiguriert

Lösung:

```
# 1. Prüfe Ingress
kubectl get ingress -n development

# 2. Prüfe Ingress Controller
kubectl get pods -n ingress-nginx

# Falls nicht installiert:
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/main/deploy/static/prov
```

```
# 3. Alternative: Port-Forward direkt zur App
kubectl port-forward svc/manifest-app-service -n development 5000:80

# Dann Browser: http://localhost:5000/swagger
```

Problem: ArgoCD zeigt "OutOfSync"

Ursache: Git und Cluster stimmen nicht überein

Lösung:

```
# In ArgoCD UI:
# 1. Klicke auf die App
# 2. Klicke "SYNC"
# 3. Klicke "SYNCHRONIZE"

# ODER im Terminal:
kubectl annotate application manifest-app -n argocd \
  argocd.argoproj.io/refresh=hard --overwrite
```

🔧 Nützliche Befehle

Status prüfen

```
# Alle Pods sehen
kubectl get pods -n development

# Alle Services sehen
kubectl get svc -n development

# Alles auf einmal
kubectl get all -n development

# ArgoCD Status
kubectl get application -n argocd
```

Logs ansehen

```
# Logs eines Pods
kubectl logs -n development <POD-NAME>

# Logs folgen (live)
kubectl logs -n development <POD-NAME> -f

# Logs aller Pods mit Label
kubectl logs -n development -l app=manifest-app --tail=50
```

In Container einsteigen (Debugging)

```
# Shell in Pod öffnen
kubectl exec -it -n development <POD-NAME> -- /bin/sh

# Befehle im Container:
# - ls /app                    (Dateien ansehen)
# - printenv                   (Umgebungsvariablen)
```

```
# - cat /etc/app-secrets/db_user   (Secret-Dateien)
# - exit                       (Container verlassen)
```

Aufräumen

```
# Nur Pods neu starten
kubectl rollout restart deployment/manifest-app -n development

# Application löschen (Daten bleiben!)
kubectl delete application manifest-app -n argocd

# Alles löschen (inkl. Daten!)
kubectl delete namespace development
```

🆘 Schnelle Hilfe

"Ich habe meine Daten verloren!"

```
# 1. Prüfe verfügbare Backups
.\restore-backup.ps1 -ListBackups

# 2. Restore das neueste
.\restore-backup.ps1 -BackupFile continuous-XXXXXX-XXXXXX.sql
```

"ArgoCD zeigt Fehler!"

```
# App-Details ansehen
kubectl get application manifest-app -n argocd -o yaml

# App löschen & neu erstellen
kubectl delete application manifest-app -n argocd
kubectl apply -f argocd/application.yaml
```

"Ich will neu anfangen!"

```
# 1. BACKUP MACHEN!
.\backup-now.ps1 -BackupName "before-reset" -Download

# 2. Alles löschen
kubectl delete namespace development
kubectl delete application manifest-app -n argocd

# 3. Neu deployen
kubectl apply -f argocd/application.yaml

# 4. ImagePullSecret neu erstellen
kubectl create namespace development
kubectl create secret docker-registry harbor-regcred \
  --docker-server=localhost:30002 \
  --docker-username=admin \
  --docker-password=Harbor12345 \
  --docker-email=admin@local \
  -n development

# 5. Warten & prüfen
kubectl get pods -n development -w
```

📚 Weiterführende Ressourcen

- **Kubernetes Basics:** https://kubernetes.io/de/docs/tutorials/
- **ArgoCD Docs:** https://argo-cd.readthedocs.io/
- **Harbor Docs:** https://goharbor.io/docs/
- **Swagger UI:** `http://localhost/swagger` (deine eigene!)

✅ Checkliste "Alles läuft"

Wenn du alle folgenden Punkte abhaken kannst, läuft alles perfekt:

```
□ Harbor erreichbar (localhost:30002)
□ Image in Harbor sichtbar (studenten/manifest-app:latest)
□ ArgoCD erreichbar (localhost:8080)
□ Application in ArgoCD: Status "Synced", Health "Healthy"
□ kubectl get pods -n development → alle Running (1/1)
□ kubectl get pvc -n development → 3 PVCs Bound
□ Swagger UI öffnet (localhost/swagger)
□ POST /api/student funktioniert
□ GET /api/student zeigt Daten
□ Backup-Daemon läuft (kubectl logs -n development -l app=postgres-backup)
```

Wenn ALLES ✓ → Du bist ein Kubernetes-Profi! 🎓

🚨 Wichtige Sicherheitshinweise

Für Production (später):

1. ❌ **NIEMALS** `secrets/` Ordner committen!
   → Steht schon in `.gitignore`, trotzdem aufpassen
2. ❌ **NIEMALS** Passwörter in YAML-Dateien hardcoden
   → Nutze Secrets (machst du schon ✓)
3. ✅ **IMMER** vor großen Änderungen: `.\backup-now.ps1 -Download`
4. ✅ **Teste Restores** regelmäßig (1x/Monat):
   → Backup ist nutzlos, wenn Restore nicht funktioniert!
5. ✅ **Überwache Backups:**
   `powershell # Letztes Backup-Datum prüfen $pod = kubectl get pods -n development -l app=postgres-backup -o jsonpath="{.items[0].metadata.name}" kubectl exec -n development $pod -- ls -lt /backups/ | head -n 2`

Viel Erfolg mit deinem Deployment! 🚀

Bei Fragen: Lies die README-Dateien in `k8s/postgres/` oder schau in die ArgoCD UI.

==========

YAML-EXAMPLES.md

==========

📝 Kommentierte YAML-Dateien - Jede Zeile erklärt

Inhaltsverzeichnis

---

1. Namespace

Was ist das?
Ein "Ordner" in Kubernetes. Alle deine Ressourcen (Pods, Services, etc.) liegen darin.

Warum brauchst du das?
Trennung von verschiedenen Projekten (z.B. `development`, `production`).

```
# k8s/Student-api/Namespace.yaml

# API-Version: Welche Kubernetes-Version diese Ressource unterstützt
apiVersion: v1

# Art der Ressource: Ein Namespace ist ein "Ordner" für andere Ressourcen
kind: Namespace

# Metadaten: Informationen ÜBER die Ressource
metadata:
  # Name des Namespace - WICHTIG: alle anderen Ressourcen müssen diesen Namen nutzen
  name: development

  # Labels: Markierungen zum Filtern und Organisieren (optional)
  labels:
    environment: dev          # Zeigt: das ist die Entwicklungsumgebung
    managed-by: argocd        # Zeigt: ArgoCD verwaltet diesen Namespace
```

---

2. ConfigMap

Was ist das?
Speichert **nicht-geheime** Einstellungen (wie DB-Name, Hostnamen).

Warum nicht direkt im Code?
Du kannst Einstellungen ändern, ohne die App neu zu bauen!

```
# k8s/Student-api/ConfigMap.yaml

apiVersion: v1

# Art der Ressource: ConfigMap = Konfigurations-Speicher
kind: ConfigMap

metadata:
  # Name der ConfigMap - wird später in Deployment.yaml referenziert
  name: app-config

  # In welchem Namespace liegt diese ConfigMap?
  namespace: development

# data: Die eigentlichen Konfigurations-Daten (Key-Value Paare)
data:
  # Key: database-host, Value: postgres-service
  # Die App liest das später über Umgebungsvariablen
  database-host: "postgres-service"   # Name des Postgres-Service

  # Name der Datenbank
  database-name: "studentdb"
```

Wie nutzt die App das?
In `Deployment.yaml` wird das so gemappt:

```
env:
- name: DB_HOST                    # Name der Umgebungsvariable in der App
  valueFrom:
    configMapKeyRef:
      name: app-config              # Name der ConfigMap (oben definiert)
      key: database-host            # Welcher Key aus der ConfigMap
```

---

3. Secret

Was ist das?
Speichert **geheime** Daten (Passwörter, API-Keys). Wird verschlüsselt gespeichert.

Unterschied zu ConfigMap?
Secrets sind Base64-kodiert und haben spezielle Berechtigungen.

```
# k8s/Student-api/Secret.yaml (BEISPIEL - in deinem Projekt anders!)

apiVersion: v1

# Art der Ressource: Secret = geheime Daten
kind: Secret

metadata:
  name: db-credentials          # Name des Secrets
  namespace: development

# type: Art des Secrets
# Opaque = generisches Secret (am häufigsten)
# kubernetes.io/dockerconfigjson = für ImagePullSecrets
type: Opaque
```

```
# stringData: Daten im Klartext (Kubernetes kodiert automatisch zu Base64)
stringData:
  # Username für die Datenbank
  username: "app_user"

  # Passwort für die Datenbank
  password: "SuperSecurePassword123!"

# ALTERNATIV: data (schon Base64-kodiert)
# data:
#   username: YXBwX3VzZXI=                    # Base64 von "app_user"
#   password: U3VwZXJTZWN1cmVQYXNzd29yZDEyMyE=  # Base64 von "SuperSecure..."
```

Wie erstelle ich Base64?

```
# PowerShell
[Convert]::ToBase64String([System.Text.Encoding]::UTF8.GetBytes("mein-text"))

# Bash/Linux
echo -n "mein-text" | base64
```

Wie nutzt die App das?

```
env:
- name: POSTGRES_PASSWORD        # Name der Umgebungsvariable
  valueFrom:
    secretKeyRef:
      name: db-credentials        # Name des Secrets
      key: password               # Welcher Key aus dem Secret
```

---

4. Deployment

Was ist das?
Die wichtigste Datei! Sagt Kubernetes:
- Welches Image starten?
- Wie viele Kopien (Replicas)?
- Welche Ressourcen (CPU/RAM)?
- Wie prüfen, ob die App läuft?

```
# k8s/Student-api/Deployment.yaml

# API-Version für Deployments
apiVersion: apps/v1

# Art der Ressource: Deployment = verwaltet Pods
kind: Deployment

metadata:
  # Name des Deployments - wird in Befehlen genutzt
  # z.B.: kubectl get deployment manifest-app
  name: manifest-app

  namespace: development

  # Labels für das Deployment selbst
  labels:
```

```
    app: manifest-app          # Haupt-Label (wichtig!)
    version: v1                # Versionierung
    component: api             # Art der Komponente

# spec: Die Spezifikation - WIE soll das Deployment aussehen?
spec:
  # replicas: Wie viele Kopien der App sollen laufen?
  # 2 = Hochverfügbarkeit (wenn eine abstürzt, läuft die andere)
  replicas: 2

  # strategy: Wie sollen Updates ablaufen?
  strategy:
    type: RollingUpdate        # Schrittweise ersetzen (kein Downtime)
    rollingUpdate:
      maxSurge: 1              # Maximal 1 Pod mehr als replicas (2+1=3 während Update)
      maxUnavailable: 0        # Mindestens 2 müssen immer laufen (zero-downtime)

  # selector: Wie findet Kubernetes die zugehörigen Pods?
  # MUSS mit labels der Pods übereinstimmen!
  selector:
    matchLabels:
      app: manifest-app        # Suche alle Pods mit diesem Label

  # template: Die "Vorlage" für jeden Pod
  template:
    # Metadaten für die Pods (nicht für das Deployment!)
    metadata:
      labels:
        app: manifest-app      # MUSS mit selector.matchLabels übereinstimmen!
        version: v1
        component: api

      # annotations: Zusätzliche Metadaten (nicht für Selektion)
      annotations:
        # Prometheus-Monitoring (falls installiert)
        prometheus.io/scrape: "true"   # Dieser Pod soll überwacht werden
        prometheus.io/port: "8080"     # Port für Metriken
        prometheus.io/path: "/metrics" # Pfad zu Metriken

    # spec: WIE soll der Pod aussehen?
    spec:
      # securityContext: Sicherheitseinstellungen für den ganzen Pod
      securityContext:
        runAsNonRoot: true     # Container NICHT als Root laufen lassen
        runAsUser: 1000        # Nutzer-ID im Container
        fsGroup: 1000          # Gruppen-ID für Dateisystem-Zugriff

      # imagePullSecrets: Passwort für private Docker-Registry (Harbor)
      imagePullSecrets:
        - name: harbor-regcred # Name des Secrets (kubectl create secret docker-registry...)

      # initContainers: Container, die VOR der App starten
      # Nutzen: Warten bis Datenbank bereit ist
      initContainers:
      - name: wait-for-postgres    # Name des Init-Containers
        image: postgres:15-alpine  # Nutzt Postgres-Image (hat pg_isready)

        # command: Was soll dieser Container tun?
        command:
          - sh                # Shell starten
          - -c                # Führe folgenden Befehl aus
          - >                 # Mehrzeiligen Befehl (YAML-Syntax)
            until pg_isready -h postgres-service -U "$POSTGRES_USER" -d "$POSTGRES_DB";
```

```
          do echo "waiting for postgres"; sleep 2; done
      # Bedeutung: Wiederhole pg_isready bis Postgres antwortet

      # env: Umgebungsvariablen für den Init-Container
      env:
      - name: POSTGRES_DB
        valueFrom:
          configMapKeyRef:
            name: app-config
            key: database-name

      - name: POSTGRES_USER
        valueFrom:
          secretKeyRef:
            name: db-credentials
            key: username

      - name: POSTGRES_PASSWORD
        valueFrom:
          secretKeyRef:
            name: db-credentials
            key: password

      # PGPASSWORD: Spezielle Variable für PostgreSQL-Tools
      - name: PGPASSWORD
        valueFrom:
          secretKeyRef:
            name: db-credentials
            key: password

    # containers: Die Haupt-Container (deine App!)
    containers:
    - name: manifest-app         # Name des Containers

      # image: Welches Docker-Image starten?
      # Format: registry/project/image:tag
      image: localhost:30002/studenten/manifest-app:latest

      # imagePullPolicy: Wann Image neu herunterladen?
      # Always = bei jedem Start (gut für :latest Tag)
      # IfNotPresent = nur wenn nicht lokal vorhanden
      # Never = nie herunterladen (nur lokale Images)
      imagePullPolicy: Always

      # ports: Welche Ports öffnet der Container?
      ports:
      - name: http           # Name des Ports (frei wählbar)
        containerPort: 8080  # Port INNERHALB des Containers
        protocol: TCP        # TCP oder UDP

      # resources: Wie viel CPU/RAM bekommt der Container?
      resources:
        # requests: Garantierte Ressourcen (Minimum)
        requests:
          cpu: 100m          # 100 Millicores = 0.1 CPU-Core
          memory: 128Mi      # 128 Megabyte RAM

        # limits: Maximale Ressourcen (Container wird gedrosselt/beendet)
        limits:
          cpu: 500m          # 500 Millicores = 0.5 CPU-Core
          memory: 512Mi      # 512 Megabyte RAM

      # securityContext: Sicherheit für diesen Container
```

```
        securityContext:
          allowPrivilegeEscalation: false  # Keine Rechte-Erweiterung
          readOnlyRootFilesystem: false    # Root-Dateisystem beschreibbar (App braucht das)
          capabilities:
            drop:
            - ALL              # Entferne alle Linux-Capabilities (Rechte)

        # startupProbe: Ist die App fertig gestartet?
        # Wird nur EINMAL beim Start geprüft
        startupProbe:
          httpGet:
            path: /api/student     # Welcher Pfad soll geprüft werden?
            port: 8080             # Auf welchem Port?
          initialDelaySeconds: 10  # Warte 10 Sek nach Start
          periodSeconds: 5         # Prüfe alle 5 Sek
          timeoutSeconds: 3        # Antwort muss in 3 Sek kommen
          failureThreshold: 10     # 10 Fehlversuche = Pod neustart (10*5=50s max)

        # livenessProbe: Läuft die App NOCH?
        # Wenn das fehlschlägt: Pod wird neugestartet
        livenessProbe:
          httpGet:
            path: /api/student
            port: 8080
          initialDelaySeconds: 30  # Warte 30 Sek nach Start
          periodSeconds: 10        # Prüfe alle 10 Sek
          timeoutSeconds: 3
          failureThreshold: 3      # 3 Fehlversuche = Neustart

        # readinessProbe: Ist die App bereit für Traffic?
        # Wenn das fehlschlägt: Pod bekommt KEINEN Traffic vom Service
        readinessProbe:
          httpGet:
            path: /api/student
            port: 8080
          initialDelaySeconds: 5   # Prüfe früh
          periodSeconds: 5         # Prüfe oft
          timeoutSeconds: 3
          failureThreshold: 3      # 3 Fehlversuche = aus Loadbalancer entfernen

        # env: Umgebungsvariablen für die App
        env:
        # DB_HOST wird aus ConfigMap gelesen
        - name: DB_HOST
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: database-host

        # DB_NAME wird aus ConfigMap gelesen
        - name: DB_NAME
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: database-name

        # volumeMounts: Verzeichnisse, die in den Container gemountet werden
        volumeMounts:
        - name: secret-volume      # Name muss mit volumes.name übereinstimmen
          mountPath: /etc/app-secrets  # Pfad IM Container
          readOnly: true           # Nur lesen, nicht schreiben

      # volumes: Definiert, WOHER die Daten kommen
```

```
      volumes:
      - name: secret-volume        # Name für volumeMounts
        secret:
          secretName: db-credentials  # Welches Secret?
          items:                  # Welche Keys als Dateien?
          - key: password         # Key aus Secret
            path: db_password     # Dateiname: /etc/app-secrets/db_password
          - key: username
            path: db_user         # Dateiname: /etc/app-secrets/db_user

---
# Service für das Deployment
# Ein Service ist eine "feste Adresse" für Pods

apiVersion: v1
kind: Service

metadata:
  name: manifest-app-service  # Name des Service (wird von Ingress genutzt)
  namespace: development
  labels:
    app: manifest-app
    component: api

spec:
  # type: Art des Service
  # ClusterIP = nur INNERHALB des Clusters erreichbar (Standard)
  # NodePort = auch von außen erreichbar auf Port 30000-32767
  # LoadBalancer = Cloud-Loadbalancer (AWS ELB, Azure LB, etc.)
  type: ClusterIP

  # selector: Welche Pods gehören zu diesem Service?
  # MUSS mit Pod-Labels übereinstimmen!
  selector:
    app: manifest-app

  # ports: Welche Ports werden weitergeleitet?
  ports:
  - name: http           # Name des Ports (frei wählbar)
    port: 80             # Port des SERVICE (ClusterIP:80)
    targetPort: 8080     # Port des PODS (Container läuft auf 8080)
    protocol: TCP

  # Bedeutung:
  # Traffic auf Service-Port 80 → wird zu Pod-Port 8080 weitergeleitet
```

---

5. Service

Was ist das?
Eine "feste Adresse" für deine Pods. Pods haben wechselnde IPs, der Service hat eine stabile IP.

Warum brauchst du das?
Andere Pods/Services können deine App erreichen, egal welcher Pod gerade läuft.

```
# k8s/Student-api/Service.yaml (normalerweise in Deployment.yaml)

apiVersion: v1
kind: Service
```

```
metadata:
  name: manifest-app-service
  namespace: development
  labels:
    app: manifest-app

spec:
  # type: Wie ist der Service erreichbar?
  type: ClusterIP        # Nur innerhalb Kubernetes (Standard)
  # Alternativen:
  # - NodePort: Auch von außen auf Node-IP:30000-32767
  # - LoadBalancer: Cloud-Loadbalancer (AWS, Azure, GCP)

  # selector: Welche Pods gehören zu diesem Service?
  selector:
    app: manifest-app    # Suche Pods mit diesem Label

  # ports: Port-Mapping
  ports:
  - name: http
    port: 80             # Externer Port (andere Services nutzen diesen)
    targetPort: 8080     # Interner Port (Pod lauscht auf diesem)
    protocol: TCP

# Ergebnis:
# Andere Pods können erreichen via:
# - http://manifest-app-service (innerhalb des Namespace)
# - http://manifest-app-service.development (namespace-übergreifend)
# - http://manifest-app-service.development.svc.cluster.local (voller DNS)
```

Service-Typen im Detail:

Typ	Erreichbar von	Use Case	Port-Range
ClusterIP	Nur innerhalb Kubernetes	Standard, für interne Services	beliebig
NodePort	Außen über Node-IP	Entwicklung, Testing	30000-32767
LoadBalancer	Außen über Cloud-LB	Produktion in Cloud	beliebig

---

6. Ingress

Was ist das?
Der "Türsteher" - leitet HTTP/HTTPS-Traffic von außen zu deinen Services.

Warum brauchst du das?
Damit du im Browser `http://localhost/swagger` öffnen kannst.

```
# k8s/Student-api/Ingress.yaml

# API-Version für Ingress
apiVersion: networking.k8s.io/v1

# Art der Ressource: Ingress = HTTP-Router
kind: Ingress

metadata:
```

```
  name: manifest-app-ingress
  namespace: development

  # annotations: Konfiguration für den Ingress Controller
  annotations:
    # Welcher Ingress Controller soll das verarbeiten?
    # nginx = NGINX Ingress Controller (am häufigsten)
    kubernetes.io/ingress.class: "nginx"

    # nginx-spezifische Einstellungen
    nginx.ingress.kubernetes.io/rewrite-target: /
    # Bedeutung: /api/student → /api/student (keine Änderung)

    # SSL/TLS (HTTPS) - optional
    # nginx.ingress.kubernetes.io/ssl-redirect: "false"  # Kein HTTPS-Zwang

spec:
  # ingressClassName: Moderne Alternative zu annotations
  ingressClassName: nginx

  # rules: Routing-Regeln
  rules:
  # host: Für welchen Hostnamen gilt diese Regel?
  - host: localhost      # Nur für localhost
    # Alternativ:
    # - host: api.example.com
    # - host: "*.example.com"  (Wildcard)

    http:
      # paths: Welche Pfade werden geroutet?
      paths:
      - path: /          # Alle Pfade ab Root

        # pathType: Wie soll der Pfad interpretiert werden?
        # Prefix = alles was mit /api beginnt
        # Exact = nur exakt /api
        # ImplementationSpecific = Controller entscheidet
        pathType: Prefix

        # backend: Wohin soll der Traffic gehen?
        backend:
          service:
            name: manifest-app-service  # Name des Service
            port:
              number: 80               # Port des Service

# Ergebnis:
# http://localhost/ → manifest-app-service:80 → Pod:8080
# http://localhost/swagger → manifest-app-service:80 → Pod:8080/swagger
# http://localhost/api/student → manifest-app-service:80 → Pod:8080/api/student
```

Ingress vs. Service:

	Service	Ingress
Layer	Layer 4 (TCP/UDP)	Layer 7 (HTTP/HTTPS)
Routing	Nach IP/Port	Nach Host/Pfad

	Service	Ingress
SSL/TLS	Nein	Ja
Loadbalancing	Round-Robin	Konfigurierbar

Beispiel-Szenarien:

```
# Szenario 1: Mehrere Services auf einem Host
rules:
- host: localhost
  http:
    paths:
    - path: /api
      pathType: Prefix
      backend:
        service:
          name: api-service
          port:
            number: 80
    - path: /frontend
      pathType: Prefix
      backend:
        service:
          name: frontend-service
          port:
            number: 80

# Szenario 2: Mehrere Hosts
rules:
- host: api.example.com
  http:
    paths:
    - path: /
      pathType: Prefix
      backend:
        service:
          name: api-service
          port:
            number: 80
- host: app.example.com
  http:
    paths:
    - path: /
      pathType: Prefix
      backend:
        service:
          name: frontend-service
          port:
            number: 80
```

---

7. PersistentVolumeClaim

Was ist das?
Eine "Anfrage" für Speicherplatz. Wie eine externe Festplatte für Pods.

Warum brauchst du das?
Container sind "ephemeral" (flüchtig) - Daten verschwinden bei Neustart. PVCs bleiben!

```
# k8s/postgres/persistent-volumes.yaml

apiVersion: v1

# Art der Ressource: PersistentVolumeClaim = Speicher-Anfrage
kind: PersistentVolumeClaim

metadata:
  name: postgres-data-postgres-0  # Name des PVC
  namespace: development

  # annotations: Spezielle Anweisungen für ArgoCD
  annotations:
    # Prune=false = ArgoCD darf diesen PVC NICHT löschen!
    # Wichtig: Schützt Datenbank-Daten vor versehentlichem Löschen
    argocd.argoproj.io/sync-options: Prune=false

spec:
  # accessModes: Wie darf auf den Speicher zugegriffen werden?
  accessModes:
  - ReadWriteOnce     # RWO = Ein Pod kann lesen+schreiben
  # Alternativen:
  # - ReadOnlyMany    # ROX = Viele Pods können lesen
  # - ReadWriteMany   # RWX = Viele Pods können lesen+schreiben (NFS, etc.)

  # storageClassName: Welche Art von Speicher?
  # hostpath = Lokaler Ordner auf dem Node (Docker Desktop)
  # standard = Cloud-Standard (AWS EBS, Azure Disk)
  # fast = SSD-Storage
  storageClassName: hostpath

  # resources: Wie viel Speicher?
  resources:
    requests:
      storage: 10Gi   # 10 Gigabyte

# Nach Erstellung wird automatisch ein PersistentVolume (PV) erstellt und "gebunden"
```

PVC Lifecycle:

```
1. PVC erstellt → Status: Pending
2. Kubernetes findet/erstellt PV → Status: Bound
3. Pod nutzt PVC → Daten werden geschrieben
4. Pod gelöscht → PVC bleibt (Status: Bound)
5. PVC gelöscht → PV bleibt (wenn ReclaimPolicy: Retain)
```

Nutzung in Pod:

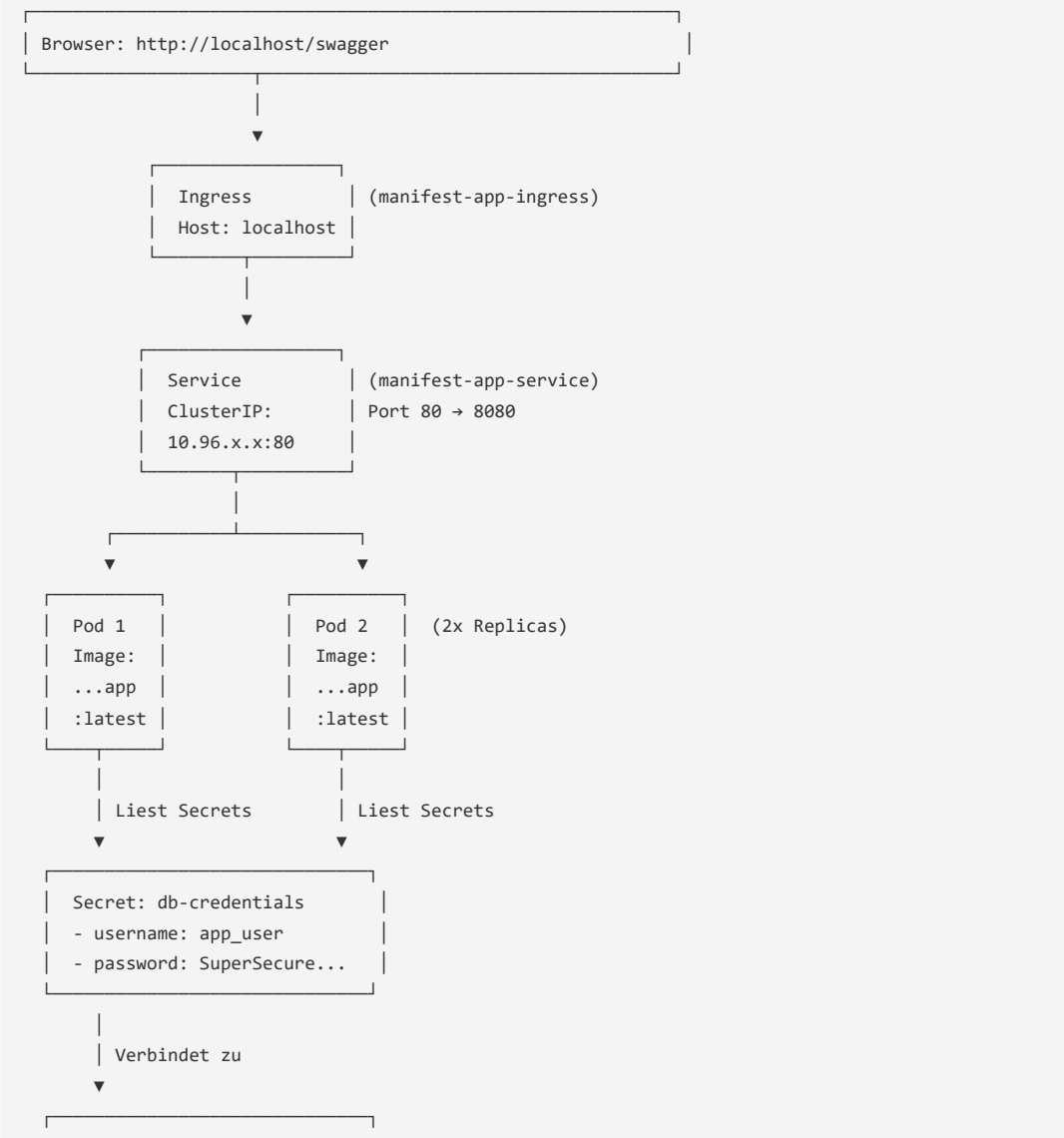
```
# In Deployment/StatefulSet
spec:
  template:
    spec:
      containers:
      - name: postgres
        volumeMounts:
        - name: data              # Name (frei wählbar)
          mountPath: /var/lib/postgresql/data  # Pfad im Container

      volumes:
      - name: data                # Muss mit volumeMounts.name übereinstimmen
```

```
        persistentVolumeClaim:
          claimName: postgres-data-postgres-0  # Name des PVC
```

Prüfen:

```
# PVCs ansehen
kubectl get pvc -n development

# Output:
# NAME                        STATUS   VOLUME                                     CAPACITY   STORAG
# postgres-data-postgres-0    Bound    pvc-xxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx       10Gi       hostpa

# PVs ansehen
kubectl get pv

# Details
kubectl describe pvc postgres-data-postgres-0 -n development
```

Zusammenfassung: Wie hängt alles zusammen?



```
  │  Service: postgres-service    │ Port 5432
  └──────────┬──────────────────────┘
             │
             ▼
     ┌──────────────┐
     │  Pod:        │
     │  postgres-0  │ (StatefulSet)
     └──────┬───────┘
            │
            │ Schreibt/Liest
            ▼
    ┌──────────────────────┐
    │  PVC:                │
    │  postgres-data-      │ 10Gi Storage
    │  postgres-0          │
    └──────────────────────┘
```

Quick Reference: Häufige YAML-Patterns

Pattern 1: Umgebungsvariable aus ConfigMap

```yaml
env:
- name: MY_VAR
  valueFrom:
    configMapKeyRef:
      name: my-config
      key: my-key
```

Pattern 2: Umgebungsvariable aus Secret

```yaml
env:
- name: MY_SECRET
  valueFrom:
    secretKeyRef:
      name: my-secret
      key: password
```

Pattern 3: Alle Keys aus ConfigMap als Env-Vars

```yaml
envFrom:
- configMapRef:
    name: my-config
# Ergebnis: Jeder Key wird zu einer Env-Var
```

Pattern 4: Volume aus Secret

```yaml
volumes:
- name: secret-volume
  secret:
    secretName: my-secret
containers:
- name: app
  volumeMounts:
  - name: secret-volume
    mountPath: /etc/secrets
```

```
    readOnly: true
# Ergebnis: /etc/secrets/password (Datei mit Inhalt)
```

Pattern 5: Multi-Container Pod

```
containers:
- name: app
  image: my-app:latest
- name: sidecar
  image: logging-agent:latest
# Beide Container teilen sich Netzwerk & Volumes
```

Nächste Schritte:
Lies `DEPLOYMENT-BLUEPRINT.md` für die komplette Deployment-Anleitung!

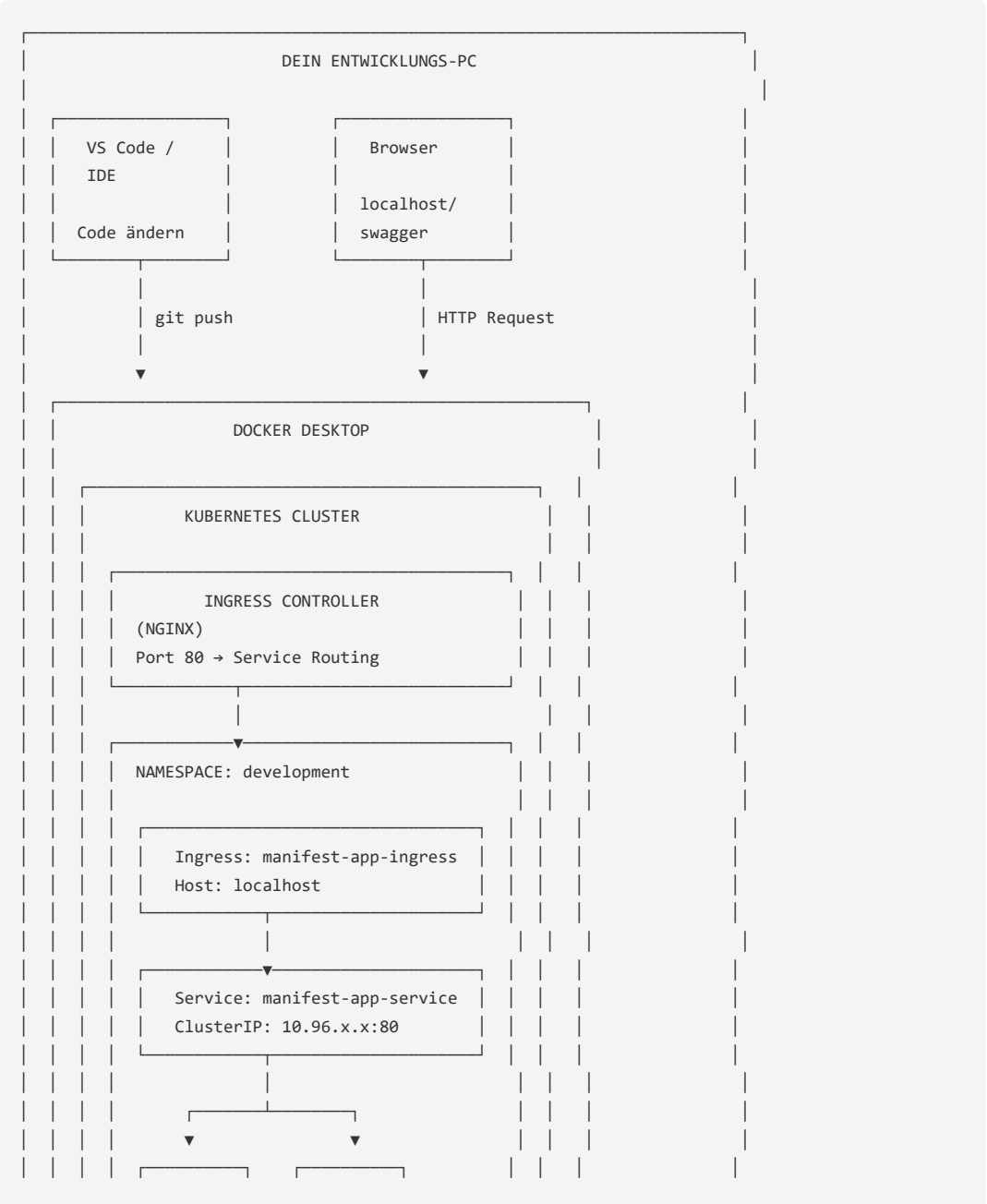
==========

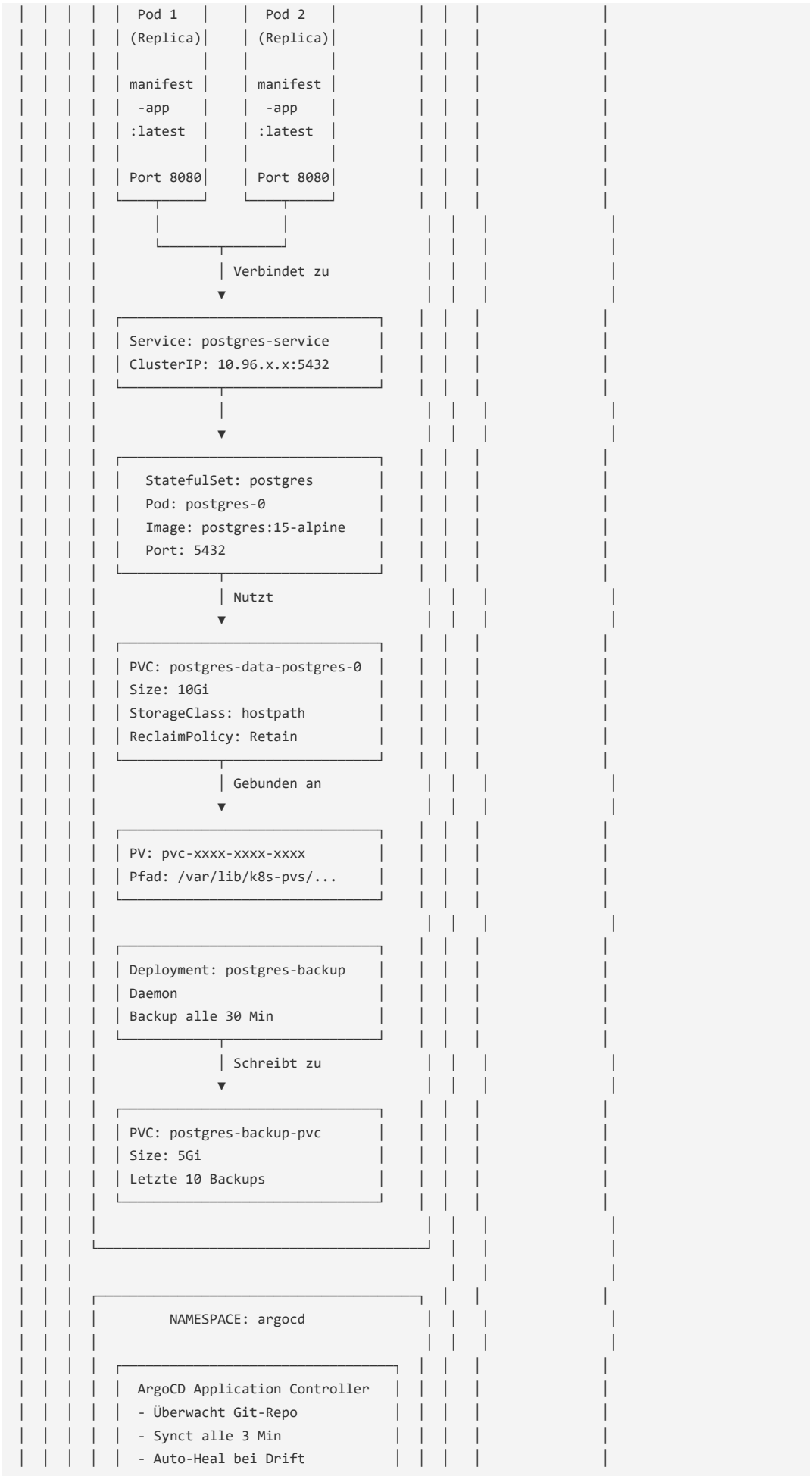
ARCHITECTURE.md

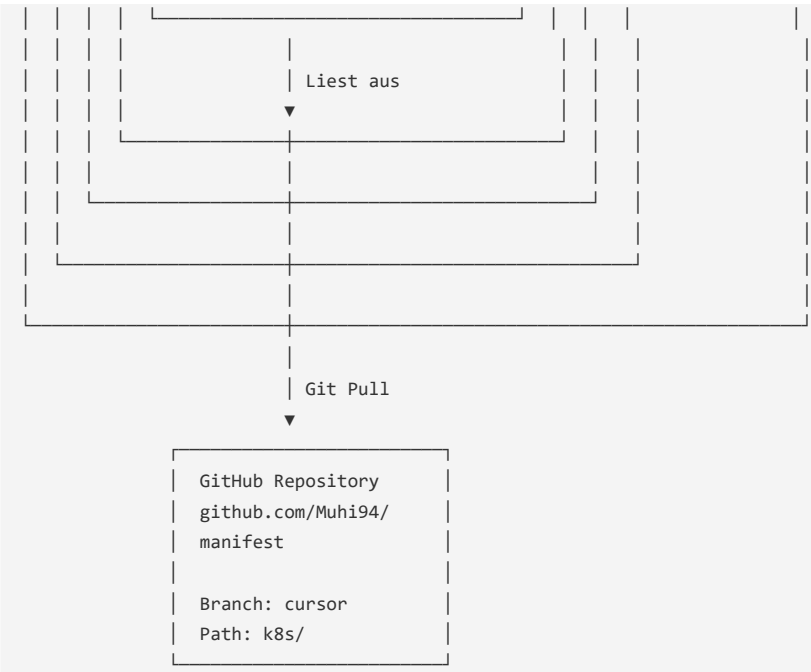
==========

📐 Architektur-Übersicht - Wie alles zusammenhängt

📐 System-Architektur




Pod 1 (Replica)
Pod 2 (Replica)
manifest -app :latest
Port 8080
Verbindet zu
Service: postgres-service
ClusterIP: 10.96.x.x:5432
StatefulSet: postgres
Pod: postgres-0
Image: postgres:15-alpine
Port: 5432
Nutzt
PVC: postgres-data-postgres-0
Size: 10Gi
StorageClass: hostpath
ReclaimPolicy: Retain
Gebunden an
PV: pvc-xxxx-xxxx-xxxx
Pfad: /var/lib/k8s-pvs/...
Deployment: postgres-backup
Daemon
Backup alle 30 Min
Schreibt zu
PVC: postgres-backup-pvc
Size: 5Gi
Letzte 10 Backups
NAMESPACE: argocd
ArgoCD Application Controller
- Überwacht Git-Repo
- Synct alle 3 Min
- Auto-Heal bei Drift



🔄 Datenfluss: HTTP-Request → Datenbank

Szenario: Benutzer ruft `GET /api/student` auf

```
1. Browser
   ↓
   GET http://localhost/api/student
   ↓
2. Docker Desktop Port Binding
   ↓
   localhost:80 → Kubernetes Cluster
   ↓
3. Ingress Controller (NGINX)
   ↓
   Host: localhost → Route zu manifest-app-ingress
   ↓
4. Ingress: manifest-app-ingress
   ↓
   Path: /api → Backend: manifest-app-service:80
   ↓
5. Service: manifest-app-service
   ↓
   ClusterIP:80 → Load Balance zwischen Pods
   ↓
6. Pod: manifest-app-xxx (einer von 2)
   ↓
   Port 8080 → ASP.NET Core App
   ↓
7. App liest Umgebungsvariablen
   ↓
   DB_HOST = postgres-service (aus ConfigMap)
   DB_NAME = studentdb (aus ConfigMap)
   DB_USER = app_user (aus Secret)
   DB_PASSWORD = Super... (aus Secret)
   ↓
8. App verbindet zu Postgres
```

```
   ↓
   postgres-service:5432
   ↓
9. Service: postgres-service
   ↓
   Route zu StatefulSet Pod
   ↓
10. Pod: postgres-0
    ↓
    Postgres-Datenbank
    ↓
    Liest von PVC: postgres-data-postgres-0
    ↓
11. Daten zurück zur App
    ↓
12. App serialisiert zu JSON
    ↓
13. Response zurück zum Browser
    ↓
14. Browser zeigt JSON:
    [{"id": 1, "name": "Max", "age": 25}]
```

Durchlaufzeit: ~50-200ms

🔐 Secrets & ConfigMaps Flow

```
┌──────────────────────────────────────────────────────┐
│  Git Repository (k8s/Student-api/ConfigMap.yaml)     │
│                                                      │
│  apiVersion: v1                                      │
│  kind: ConfigMap                                     │
│  data:                                               │
│    database-host: "postgres-service"                 │
│    database-name: "studentdb"                        │
└──────────────────────┬───────────────────────────────┘
                       │
                       │ ArgoCD synct
                       ▼
          ┌────────────────────────────┐
          │  ConfigMap: app-config     │
          │  in Namespace: development │
          └──────────────┬─────────────┘
                         │
                         │ Wird gelesen als Env-Var
                         ▼
             ┌──────────────────────┐
             │  Pod: manifest-app   │
             │                      │
             │  env:                │
             │  - DB_HOST=          │
             │    postgres-service  │
             │  - DB_NAME=          │
             │    studentdb         │
             └──────────────────────┘
```

```
┌──────────────────────────────────────────────────────┐
│  Lokal: secrets/db_password.txt                      │
│  Inhalt: SuperSecurePassword123!                     │
└──────────────────────┬───────────────────────────────┘
                       │
```

Image Build & Deployment Pipeline

```
        └──────────────┬──────────────┘
                       │
                       │ Kubernetes pullt Image
                       ▼
          ┌────────────────────────┐
          │  Pod: manifest-app-xxx │
          │                        │
          │  imagePullSecrets:     │
          │  - harbor-regcred      │
          │                        │
          │  image: localhost:30002/ │
          │    studenten/          │
          │    manifest-app:latest │
          │                        │
          │  imagePullPolicy: Always │
          └────────────────────────┘

3. kubectl rollout restart deployment/manifest-app -n development
   │
   ├─► Beendet alte Pods (nach RollingUpdate-Strategie)
   ├─► Startet neue Pods mit neuem Image
   └─► Zero-Downtime (maxUnavailable: 0)
       │
       ▼
   ┌──────────────────────────┐
   │ 2 neue Pods laufen       │
   │ mit neuem Code           │
   └──────────────────────────┘
```

GitOps Workflow mit ArgoCD

```
┌─────────────────────────────────────────────────────────────┐
│                      GITOPS WORKFLOW                        │
└─────────────────────────────────────────────────────────────┘

1. Änderung in YAML (z.B. Replicas erhöhen)
   │
   │  k8s/Student-api/Deployment.yaml:
   │    spec:
   │      replicas: 3  # War: 2
   │
   ▼
2. Git Commit & Push
   │
   │  git add k8s/Student-api/Deployment.yaml
   │  git commit -m "Scale to 3 replicas"
   │  git push origin cursor
   │
   ▼
3. GitHub Repository
   │  Branch: cursor
   │  Commit: abc123...
   │
   ▼
4. ArgoCD Application Controller (Poll alle 3 Min)
   │
   ├─► Vergleicht Git (Desired State) mit Cluster (Actual State)
   │   │
   │   │  Desired:  replicas: 3
   │   │  Actual:   replicas: 2
```

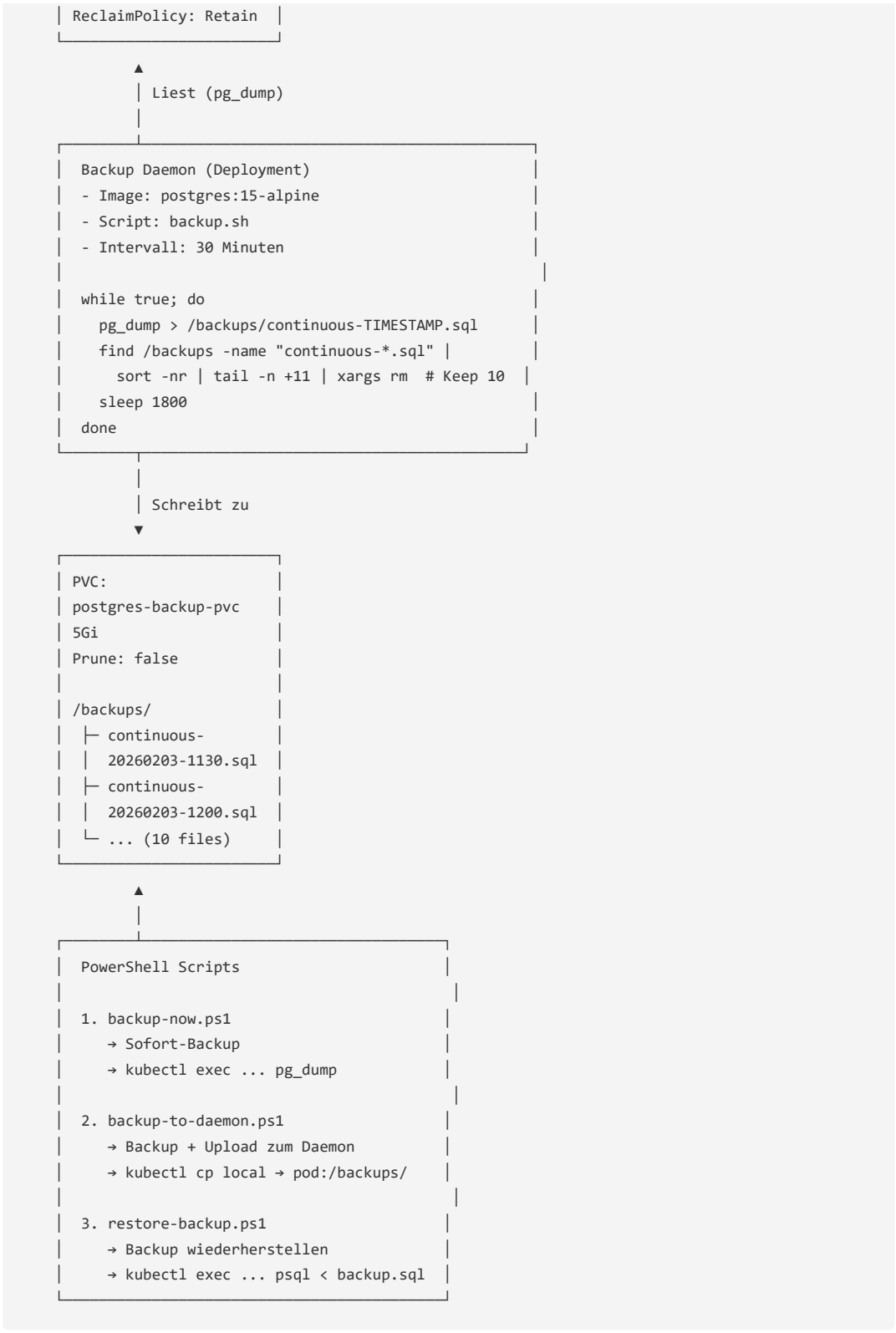
```
   │   │  → OutOfSync!
   │   │
   │   ▼
   ├─→ Auto-Sync aktiviert? → JA
   │   │
   │   ▼
   └─→ kubectl apply -f (alle geänderten Ressourcen)
       │
       ▼
5. Kubernetes Reconciliation Loop
   │
   ├─→ Deployment Controller sieht: Soll: 3, Ist: 2
   ├─→ Erstellt 1 neuen ReplicaSet
   └─→ Startet 1 neuen Pod
       │
       ▼
6. Cluster Status = Git Status
   │
   │  manifest-app-xxx-1  ✓ Running
   │  manifest-app-xxx-2  ✓ Running
   │  manifest-app-xxx-3  ✓ Running  (NEU!)
   │
   ▼
7. ArgoCD UI zeigt:
   │  Status: Synced ✓
   │  Health: Healthy ✓
   └──────────────────────────────────────────────────────────────┘
```

Self-Heal Beispiel:

```
Jemand ändert manuell:
  kubectl scale deployment/manifest-app --replicas=5 -n development

Nach 3 Minuten:
  ArgoCD sieht: Git sagt 3, Cluster hat 5 → Drift erkannt!
  → Auto-Heal: kubectl apply (setzt zurück auf 3)

Ergebnis: Git bleibt Single Source of Truth!
```

💾 Backup & Recovery Architektur

```
┌──────────────────────────────────────────────────────────────┐
│                    BACKUP SYSTEM                             │
└──────────────────────────────────────────────────────────────┘


┌──────────────────────────────────────────────────────────────┐
│  Postgres StatefulSet (postgres-0)                           │
│  - Datenbank: studentdb                                      │
│  - User: app_user                                            │
│  - Data: /var/lib/postgresql/data                            │
└──────────────┬───────────────────────────────────────────────┘
               │
               │ Schreibt/Liest
               ▼
     ┌─────────────────────────┐
     │ PVC:                    │
     │ postgres-data-          │
     │ postgres-0              │
     │ 10Gi                    │
```



Disaster Recovery Szenarien:

Szenario	PVCs Status	Daten-Status	Recovery
Pod crashed	✓ Intakt	✓ Safe	Auto-Restart
Deployment deleted	✓ Intakt	✓ Safe	Re-deploy
App deleted (ArgoCD)	✓ Intakt (Prune=false)	✓ Safe	Sync

Szenario	PVCs Status	Daten-Status	Recovery
Namespace deleted	❌ PVC deleted	✓ PV bleibt!	Rebind PV
Cluster reset	❌ Alles weg	✓ Backups in PV	Restore

🔒 Security Layers

```
┌─────────────────────────────────────────────────────────────┐
│                    SECURITY ARCHITECTURE                    │
└─────────────────────────────────────────────────────────────┘

1. Network Layer (Services)
   │
   ├─► ClusterIP Services (nicht von außen erreichbar)
   │   ├─ postgres-service:5432  → Nur innerhalb Kubernetes
   │   └─ manifest-app-service:80 → Nur über Ingress
   │
   └─► Ingress (kontrollierter Außenzugriff)
       └─ localhost → manifest-app (80 → 8080)

2. Pod Security Context
   │
   ├─► App-Pods:
   │   ├─ runAsNonRoot: true
   │   ├─ runAsUser: 1000 (nicht Root!)
   │   └─ capabilities: drop ALL
   │
   └─► Postgres-Pod:
       ├─ runAsUser: 0 (nur für hostpath notwendig)
       └─ capabilities: nur CHOWN, FSETID, FOWNER

3. Secrets Management
   │
   ├─► Git: NIEMALS Secrets committen!
   │   └─ secrets/ in .gitignore
   │
   ├─► Kubernetes Secrets (Base64-encoded)
   │   ├─ db-credentials (username/password)
   │   └─ harbor-regcred (Docker login)
   │
   └─► Zugriff:
       ├─ Als Umgebungsvariable (POSTGRES_PASSWORD)
       └─ Als Volume Mount (/etc/app-secrets/db_password)

4. RBAC (Role-Based Access Control)
   │
   └─► ArgoCD ServiceAccount
       ├─ Darf: read/write in development Namespace
       └─ Darf nicht: andere Namespaces ändern

5. Image Security
   │
   ├─► Private Registry (Harbor)
   │   └─ ImagePullSecret erforderlich
   │
   ├─► Image Scanning (Harbor-Feature)
   │   └─ CVE-Checks bei Push
   │
```

```
    └─► imagePullPolicy: Always
        └─ Immer neueste Version pullen

6. Resource Limits
   │
   └─► Verhindert DoS durch einzelne Pods:
       ├─ CPU Limit: 500m (0.5 Core)
       └─ Memory Limit: 512Mi
```

📊 Monitoring & Observability

```
┌─────────────────────────────────────────────────────────────┐
│              OBSERVABILITY STACK (optional)                 │
└─────────────────────────────────────────────────────────────┘

1. Kubernetes Events
   │
   └─► kubectl get events -n development -w
       ├─ Pod Created
       ├─ Image Pulled
       ├─ Container Started
       └─ Probe Failed

2. Pod Logs
   │
   └─► kubectl logs -n development <POD-NAME>
       ├─ Application Logs
       ├─ Error Messages
       └─ Database Connection Attempts

3. Health Checks
   │
   ├─► startupProbe → Ist App gestartet?
   ├─► livenessProbe → Läuft App noch?
   └─► readinessProbe → Bereit für Traffic?

4. ArgoCD Monitoring
   │
   └─► ArgoCD UI (localhost:8080)
       ├─ Sync Status (Synced/OutOfSync)
       ├─ Health Status (Healthy/Degraded/Progressing)
       └─ Deployment History (Rollback-fähig)

5. Prometheus (falls installiert)
   │
   └─► Annotations in Deployment:
       ├─ prometheus.io/scrape: "true"
       ├─ prometheus.io/port: "8080"
       └─ prometheus.io/path: "/metrics"
```

🔧 Technology Stack

Layer	Technology	Version	Zweck
Orchestration	Kubernetes	1.28+	Container-Orchestrierung
GitOps	ArgoCD	2.9+	Kontinuierliche Delivery

Layer	Technology	Version	Zweck
Registry	Harbor	2.9+	Image-Storage
Ingress	NGINX Ingress	1.9+	HTTP-Routing
Database	PostgreSQL	15	Relationale DB
Backend	ASP.NET Core	8.0	REST API
Language	C#	12	App-Logik
ORM	Entity Framework Core	8.0	DB-Zugriff
Storage	hostpath StorageClass	-	Persistenz (Dev)

📈 Skalierungs-Szenarien

Horizontal Scaling (Mehr Pods)

```
# k8s/Student-api/Deployment.yaml
spec:
  replicas: 5  # War: 2

# Ergebnis:
# manifest-app-xxx-1  ✓ Running
# manifest-app-xxx-2  ✓ Running
# manifest-app-xxx-3  ✓ Running (NEU)
# manifest-app-xxx-4  ✓ Running (NEU)
# manifest-app-xxx-5  ✓ Running (NEU)

# Load-Balancing automatisch durch Service!
```

Vertical Scaling (Mehr Ressourcen)

```
# k8s/Student-api/Deployment.yaml
resources:
  limits:
    cpu: 1000m     # War: 500m
    memory: 1024Mi # War: 512Mi
```

Auto-Scaling (HPA - Horizontal Pod Autoscaler)

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: manifest-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: manifest-app
  minReplicas: 2
  maxReplicas: 10
  metrics:
```

```
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70

# Bedeutung: Bei >70% CPU-Last → automatisch mehr Pods (bis 10)
```

🌐 Multi-Environment Setup (Später)

```
manifest/
├── k8s/
│   ├── base/                 # Gemeinsame Ressourcen
│   │   ├── deployment.yaml
│   │   ├── service.yaml
│   │   └── kustomization.yaml
│   │
│   ├── overlays/
│   │   ├── development/      # Dev-spezifisch
│   │   │   ├── replicas: 2
│   │   │   ├── ingress: localhost
│   │   │   └── kustomization.yaml
│   │   │
│   │   ├── staging/          # Staging-spezifisch
│   │   │   ├── replicas: 3
│   │   │   ├── ingress: staging.example.com
│   │   │   └── kustomization.yaml
│   │   │
│   │   └── production/       # Prod-spezifisch
│   │       ├── replicas: 5
│   │       ├── ingress: api.example.com
│   │       ├── resources: erhöht
│   │       └── kustomization.yaml
│   │
│   └── argocd/
│       ├── app-dev.yaml
│       ├── app-staging.yaml
│       └── app-prod.yaml
```

Nächste Schritte:
- **Quick Start:** `QUICK-START-GUIDE.md`
- **Vollständige Anleitung:** `DEPLOYMENT-BLUEPRINT.md`
- **YAML-Erklärungen:** `YAML-EXAMPLES.md`