Kubernetes Deployment Blueprint für StudentApi

Inhaltsverzeichnis

Begriffe für Anfänger

Bevor wir starten, hier die wichtigsten Begriffe einfach erklärt:

Begriff	Erklärung
Pod	Die kleinste Einheit in Kubernetes. Ein Pod ist wie ein "Container-Haus" - er enthält einen oder mehrere Container, die zusammen laufen. Stell dir vor, dein Container ist eine Person und der Pod ist die Wohnung.
Namespace	Ein "Ordner" in Kubernetes, um Ressourcen zu organisieren. Wie Abteilungen in einer Firma - z.B. `development` für Entwicklung, `production` für Live-Betrieb.
Deployment	Der "Bauplan" für deine Pods. Es sagt Kubernetes: "Ich will X Kopien meiner App laufen haben" und kümmert sich darum, dass sie immer laufen.
Service	Die "Telefonzentrale" für deine Pods. Da Pods kommen und gehen können, gibt der Service eine feste Adresse, unter der deine App erreichbar ist.
Ingress	Das "Eingangstor" von außen. Es leitet Web-Traffic von einer Domain (z.B. `meineapp.de`) zu deinem Service weiter.
ConfigMap	Ein "Notizzettel" für Einstellungen. Hier speicherst du Konfigurationen wie Datenbank-Namen, die keine Geheimnisse sind.
Secret	Ein "Tresor" für sensible Daten. Passwörter und Zugangsdaten werden hier verschlüsselt gespeichert.
Harbor	Eine "Private Garage" für Docker-Images. Wie Docker Hub, aber nur für deine Firma zugänglich.
ArgoCD	Ein "Autopilot" für Deployments. Er schaut in dein Git-Repository und deployed automatisch, wenn sich etwas ändert.

1. Ordnerstruktur

So sollte dein Repository aufgebaut sein:

```
manifest/                          # Dein Projekt-Hauptordner
│
├── StudentApi/                    # Dein .NET Anwendungscode
│   ├── Controllers/               # API-Endpunkte
│   ├── Models/                    # Datenmodelle
│   ├── Data/                      # Datenbank-Kontext
│   ├── Dockerfile                 # Bauanleitung für Docker-Image
│   ├── Program.cs                 # Startpunkt der App
│   └── appsettings.json           # App-Konfiguration
│
├── k8s/                           # ALLE Kubernetes-Dateien
│   │
│   ├── Student-api/               # Alles für deine API
│   │   ├── Namespace.yaml         # Der "Ordner" für alle Ressourcen
│   │   ├── Deployment.yaml        # Wie deine App deployed wird
│   │   ├── Service.yaml           # Wie die App intern erreichbar ist
│   │   ├── Ingress.yaml           # Wie die App von außen erreichbar ist
│   │   ├── ConfigMap.yaml         # Nicht-geheime Einstellungen
│   │   └── ImagePullSecret.yaml   # Zugang zu Harbor (NICHT committen!)
│   │
│   ├── postgres/                  # Alles für die Datenbank
│   │   ├── StatefulSet.yaml       # Datenbank-Deployment
│   │   ├── Service.yaml           # Datenbank-Erreichbarkeit
│   │   └── PVC.yaml               # Speicherplatz für Daten
│   │
│   └── sealed-secrets/            # Verschlüsselte Geheimnisse
│       ├── controller.yaml        # Der Entschlüsselungs-Dienst
│       └── db-credentials-sealed.yaml # Verschlüsselte DB-Passwörter
│
├── argocd/                        # ArgoCD Konfiguration
│   ├── application.yaml           # Sagt ArgoCD: "Deploy diese App"
│   └── sealed-secrets-controller.yaml
│
├── docker-compose.yml             # Für lokales Testen
└── pipeline.ps1                   # Build-Script für Harbor
```

Warum diese Struktur?

- **k8s/** - Alle Kubernetes-Dateien an einem Ort = ArgoCD weiß, wo es schauen muss
- **Student-api/** - App-spezifische Configs getrennt von Datenbank-Configs
- **sealed-secrets/** - Sicherheit: Verschlüsselte Secrets können sicher in Git liegen

---

2. YAML-Manifeste mit Erklärungen

2.1 Namespace (Der "Ordner")

Datei: `k8s/Student-api/Namespace.yaml`

```yaml
# ============================================================
# NAMESPACE - Erstellt einen isolierten Bereich in Kubernetes
# ============================================================
# Stell dir einen Namespace wie einen Ordner auf deinem Computer vor.
# Alle Ressourcen (Pods, Services, etc.) leben in diesem "Ordner".
# Das hilft bei der Organisation und Sicherheit.
# ============================================================

apiVersion: v1                    # Welche Kubernetes-API-Version wir nutzen (v1 = stab
kind: Namespace                   # Was wir erstellen wollen: einen Namespace
metadata:                         # "Metadaten" = Informationen ÜBER die Ressource
  name: development               # Der Name unseres Namespaces - "development" für Ent
                                  # Später könntest du auch "production" für Live-Betri
```

2.2 Deployment (Der "Bauplan")

Datei: `k8s/Student-api/Deployment.yaml`

```yaml
# ============================================================
# DEPLOYMENT - Der Bauplan für deine Anwendung
# ============================================================
# Ein Deployment sagt Kubernetes:
# "Ich möchte X Kopien meiner App laufen haben"
# Kubernetes sorgt dann dafür, dass immer X Pods laufen.
# Fällt einer aus? Kubernetes startet automatisch einen neuen!
# ============================================================

apiVersion: apps/v1               # API-Version für Deployments (apps/v1 ist Standard)
kind: Deployment                  # Wir erstellen ein Deployment
metadata:                         # Informationen über dieses Deployment
  name: student-api               # Name des Deployments (frei wählbar, aber eindeutig)
  namespace: development          # In welchem "Ordner" (Namespace) es leben soll
  labels:                         # Labels = Etiketten zum Kategorisieren
    app: student-api              # Dieses Label hilft beim Finden/Filtern

spec:                             # SPEZIFIKATION - Hier kommt das "Was und Wie"
  replicas: 2                     # ANZAHL der Pods (Kopien deiner App)
                                  # 2 = Hochverfügbarkeit - fällt einer aus, läuft der

  selector:                       # SELEKTOR - Wie findet das Deployment "seine" Pods?
    matchLabels:                  # Es sucht nach Pods mit diesem Label:
      app: student-api            # Muss mit dem Label unten bei "template" übereinstim

  template:                       # TEMPLATE - Die Vorlage für jeden Pod
    metadata:                     # Metadaten für die Pods
      labels:                     # Labels für die Pods (MUSS mit selector übereinstimm
        app: student-api          # Dieses Label verbindet Pod mit Deployment UND Servi

    spec:                         # Spezifikation für den Pod-Inhalt
      # ------------------------------------------------------
      # IMAGE PULL SECRET - Zugang zu deiner privaten Registry (Harbor)
      # ------------------------------------------------------
      imagePullSecrets:           # Liste von Secrets für Registry-Zugang
        - name: harbor-credentials # Name des Secrets (erstellen wir später!)

      containers:                 # Liste der Container im Pod (meist nur einer)
        - name: student-api       # Name des Containers (frei wählbar)

          # ------------------------------------------------------
          # IMAGE - Welches Docker-Image soll gestartet werden?
          # ------------------------------------------------------
          image: dein-harbor.de/studenten/manifest-app:v1
          # ↑ Aufbau: REGISTRY/PROJEKT/IMAGE-NAME:TAG
          # - dein-harbor.de      = Deine Harbor-Adresse
          # - studenten           = Dein Projekt in Harbor
          # - manifest-app        = Name deines Images
          # - v1                  = Version/Tag des Images

          imagePullPolicy: Always # IMMER das neueste Image holen (nicht aus Cache)
```

```
                          # Alternativen: IfNotPresent, Never

        # ------------------------------------------------------
        # PORTS - Welche Ports stellt der Container bereit?
        # ------------------------------------------------------
        ports:
          - containerPort: 8080 # Der Port, auf dem deine .NET App läuft
                          # (Definiert im Dockerfile mit EXPOSE 8080)

        # ------------------------------------------------------
        # UMGEBUNGSVARIABLEN - Einstellungen für die App
        # ------------------------------------------------------
        env:
          - name: DB_HOST       # Name der Variable (so rufst du sie im Code ab)
            valueFrom:          # Wert kommt VON woanders (nicht direkt hier)
              configMapKeyRef:  # ... und zwar aus einer ConfigMap
                name: app-config      # Name der ConfigMap
                key: database-host    # Welcher Schlüssel in der ConfigMap

          - name: DB_NAME       # Datenbank-Name
            valueFrom:
              configMapKeyRef:
                name: app-config
                key: database-name

          - name: DB_USER       # Datenbank-Benutzer (GEHEIM!)
            valueFrom:
              secretKeyRef:     # Kommt aus einem Secret (verschlüsselt)
                name: db-credentials  # Name des Secrets
                key: username         # Welcher Schlüssel im Secret

          - name: DB_PASSWORD   # Datenbank-Passwort (GEHEIM!)
            valueFrom:
              secretKeyRef:
                name: db-credentials
                key: password

        # ------------------------------------------------------
        # VOLUME MOUNTS - Dateien/Ordner in den Container einbinden
        # ------------------------------------------------------
        volumeMounts:
          - name: secrets-volume      # Name des Volumes (muss unten definiert sein)
            mountPath: /etc/app-secrets # Wohin im Container mounten
            readOnly: true            # Nur lesen, nicht schreiben

        # ------------------------------------------------------
        # RESOURCES - CPU und RAM Limits
        # ------------------------------------------------------
        # WICHTIG: Ohne Limits könnte ein Pod den ganzen Server lahmlegen!
        resources:
          requests:             # MINDESTENS so viel braucht der Container
            memory: "128Mi"     # 128 Megabyte RAM
```

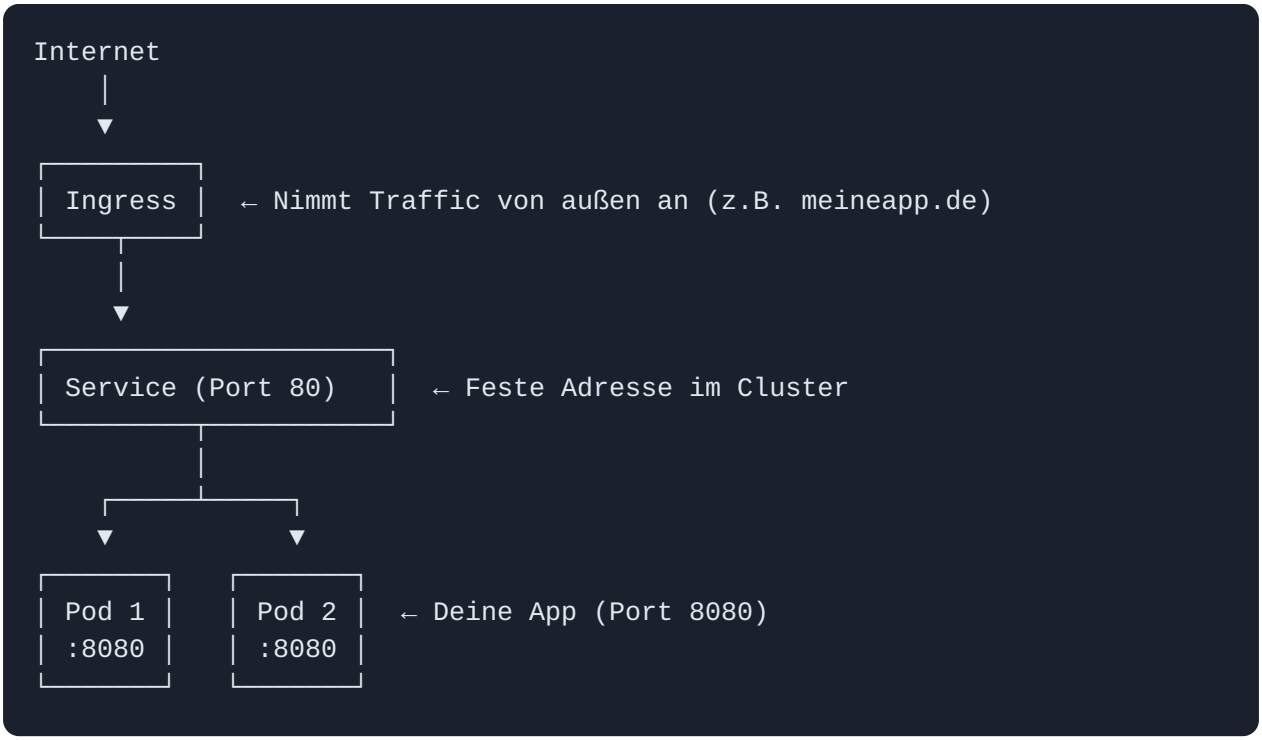
```
              cpu: "100m"         # 100 Milli-CPU (= 0.1 CPU-Kerne)

            limits:               # MAXIMAL so viel darf er nutzen
              memory: "512Mi"     # 512 Megabyte RAM
              cpu: "500m"         # 500 Milli-CPU (= 0.5 CPU-Kerne)

          # -------------------------------------------------------
          # LIVENESS PROBE - "Lebt die App noch?"
          # -------------------------------------------------------
          # Kubernetes prüft regelmäßig, ob die App antwortet.
          # Wenn nicht → Container wird neu gestartet!
          livenessProbe:
            httpGet:              # Prüfung per HTTP-Request
              path: /api/student  # Welchen Endpunkt aufrufen
              port: 8080          # Auf welchem Port
            initialDelaySeconds: 30  # Warte 30 Sek. nach Start (App braucht Zeit zum H
            periodSeconds: 10        # Dann alle 10 Sek. prüfen

          # -------------------------------------------------------
          # READINESS PROBE - "Ist die App bereit für Traffic?"
          # -------------------------------------------------------
          # Erst wenn diese Prüfung erfolgreich ist, bekommt der Pod Traffic.
          # Verhindert, dass User auf einen noch startenden Pod geleitet werden.
          readinessProbe:
            httpGet:
              path: /api/student
              port: 8080
            initialDelaySeconds: 5   # Schneller als Liveness (nur "bereit" prüfen)
            periodSeconds: 5         # Alle 5 Sek. prüfen

      # -----------------------------------------------------------
      # VOLUMES - Speicher-Definition (oben gemountet)
      # -----------------------------------------------------------
      volumes:
        - name: secrets-volume    # Name (muss oben bei volumeMounts übereinstimmen)
          secret:                 # Typ: Secret-Volume
            secretName: db-credentials  # Welches Secret als Dateien bereitstellen

---
# ============================================================
# CONFIGMAP - Nicht-geheime Einstellungen
# ============================================================
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
  namespace: development
data:                             # Hier kommen die Schlüssel-Wert-Paare
  database-host: "postgres-service"  # Hostname der Datenbank (der Service-Name!)
  database-name: "studentdb"         # Name der Datenbank
```

2.3 Service (Die "Telefonzentrale")

Datei: `k8s/Student-api/Service.yaml`

```yaml
# ============================================================
# SERVICE - Macht deine App innerhalb des Clusters erreichbar
# ============================================================
# Problem: Pods bekommen zufällige IP-Adressen und können jederzeit
#          neu gestartet werden (neue IP!).
# Lösung:  Ein Service gibt eine FESTE Adresse, die immer funktioniert.
#          Er leitet Traffic automatisch zu den richtigen Pods weiter.
# ============================================================

apiVersion: v1                    # API-Version (v1 für Services)
kind: Service                     # Wir erstellen einen Service
metadata:
  name: student-api-service       # Name des Services
                                  # WICHTIG: Unter diesem Namen ist die App erreichbar!
                                  # z.B. http://student-api-service:80 (innerhalb des C
  namespace: development          # Im gleichen Namespace wie das Deployment

spec:
  # ------------------------------------------------------------
  # TYPE - Wie soll der Service erreichbar sein?
  # ------------------------------------------------------------
  type: ClusterIP                 # Nur INTERN im Cluster erreichbar
                                  # Alternativen:
                                  # - NodePort: Auf jedem Server unter einem Port errei
                                  # - LoadBalancer: Bekommt eine externe IP (Cloud-Prov
                                  # Wir nutzen ClusterIP, weil Ingress den externen Zug

  # ------------------------------------------------------------
  # SELECTOR - Welche Pods soll der Service ansprechen?
  # ------------------------------------------------------------
  selector:
    app: student-api              # Alle Pods mit diesem Label bekommen Traffic
                                  # MUSS mit dem Label im Deployment übereinstimmen!

  # ------------------------------------------------------------
  # PORTS - Port-Konfiguration
  # ------------------------------------------------------------
  ports:
    - name: http                  # Name des Ports (optional, aber hilfreich)
      protocol: TCP               # Protokoll (TCP ist Standard für HTTP)
      port: 80                    # PORT DES SERVICES - hierüber ist er erreichbar
                                  # z.B. http://student-api-service:80
      targetPort: 8080            # PORT DES CONTAINERS - wohin wird weitergeleitet
                                  # Deine App läuft auf 8080, Service nimmt auf 80 an
                                  # Service:80 → Container:8080
```

Visualisierung des Traffics:



2.4 Ingress (Das "Eingangstor")

Datei: `k8s/Student-api/Ingress.yaml`

```yaml
# ============================================================
# INGRESS - Macht deine App von AUSSEN (Internet) erreichbar
# ============================================================
# Der Ingress ist wie ein Pförtner am Eingang:
# - Er nimmt Anfragen von außen entgegen
# - Er prüft die URL/Domain
# - Er leitet zum richtigen Service weiter
#
# VORAUSSETZUNG: Ein Ingress-Controller muss installiert sein!
# (z.B. nginx-ingress - ist bei den meisten Kubernetes-Setups dabei)
# ============================================================

apiVersion: networking.k8s.io/v1  # API-Version für Ingress-Ressourcen
kind: Ingress                     # Wir erstellen einen Ingress
metadata:
  name: student-api-ingress       # Name des Ingress
  namespace: development          # Gleicher Namespace wie App und Service

  # ------------------------------------------------------------
  # ANNOTATIONS - Zusätzliche Konfiguration für den Ingress-Controller
  # ------------------------------------------------------------
  annotations:
    # Welchen Ingress-Controller nutzen?
    kubernetes.io/ingress.class: "nginx"    # Wir nutzen nginx als Ingress-Controller

    # SSL-Redirect: Automatisch auf HTTPS umleiten?
    nginx.ingress.kubernetes.io/ssl-redirect: "false"  # Erstmal aus (kein Zertifikat n

    # CORS erlauben (für Frontend-Zugriff von anderer Domain)
    nginx.ingress.kubernetes.io/enable-cors: "true"
    nginx.ingress.kubernetes.io/cors-allow-origin: "*"  # Alle Domains erlauben
                                                         # In Produktion: Nur deine Dom

    # Rate-Limiting: Schutz vor zu vielen Anfragen
    nginx.ingress.kubernetes.io/limit-rps: "100"        # Max 100 Requests pro Sekunde

spec:
  # ------------------------------------------------------------
  # RULES - Regeln: Welche URLs wohin leiten?
  # ------------------------------------------------------------
  rules:
    - host: student-api.local     # Für welche Domain gilt diese Regel?
                                  # In Produktion: deine echte Domain (z.B. api.meinefi
                                  # Zum Testen: In /etc/hosts eintragen (siehe unten)

      http:                       # HTTP-Regeln
        paths:                    # Liste von Pfad-Regeln
          - path: /               # Welcher Pfad? "/" = alles
            pathType: Prefix      # "Prefix" = alles was mit "/" beginnt
                                  # Alternative: "Exact" = nur genau dieser Pfad
```

```
            backend:              # Wohin weiterleiten?
              service:            # An einen Service
                name: student-api-service  # Name des Services (muss existieren!)
                port:
                  number: 80      # Port des Services

# ============================================================
# LOKALES TESTEN - So testest du den Ingress auf deinem Rechner:
# ============================================================
# 1. Finde die IP deines Ingress-Controllers:
#    kubectl get ingress -n development
#
# 2. Trage in /etc/hosts (Linux/Mac) oder C:\Windows\System32\drivers\etc\hosts (Window
#    192.168.x.x  student-api.local
#    (Ersetze 192.168.x.x mit der tatsächlichen IP)
#
# 3. Öffne im Browser: http://student-api.local/api/student
# ============================================================
```

---

3. Harbor ImagePullSecret erstellen

Was ist ein ImagePullSecret?

Harbor ist deine "private Garage" für Docker-Images. Kubernetes braucht einen "Schlüssel" (Credentials), um Images daraus zu holen. Diesen Schlüssel speichern wir als **Secret**.

Schritt-für-Schritt Anleitung

Schritt 1: Terminal öffnen

Öffne ein Terminal mit kubectl-Zugang (z.B. PowerShell, Git Bash, oder Linux Terminal).

Schritt 2: Secret erstellen

Kopiere diesen Befehl und ersetze die Platzhalter:

```
kubectl create secret docker-registry harbor-credentials \
  --namespace=development \
  --docker-server=DEINE-HARBOR-URL \
  --docker-username=DEIN-BENUTZERNAME \
  --docker-password=DEIN-PASSWORT \
  --docker-email=DEINE-EMAIL
```

Beispiel mit echten Werten:

```
kubectl create secret docker-registry harbor-credentials \
  --namespace=development \
  --docker-server=harbor.meinefirma.de \
  --docker-username=max.mustermann \
  --docker-password=MeinSuperGeheimesPasswort123! \
  --docker-email=max.mustermann@firma.de
```

Schritt 3: Prüfen ob es funktioniert hat

```
kubectl get secrets -n development
```

Du solltest `harbor-credentials` in der Liste sehen.

Schritt 4: Secret Details anzeigen (ohne Passwort)

```
kubectl describe secret harbor-credentials -n development
```

Alternative: Secret als YAML-Datei

Wenn du das Secret in einer Datei speichern willst (z.B. für Automatisierung):

ACHTUNG: Diese Datei enthält dein Passwort! NIEMALS in Git committen!

```
# k8s/Student-api/ImagePullSecret.yaml
# ⚠️ NICHT IN GIT COMMITTEN! In .gitignore aufnehmen!
apiVersion: v1
kind: Secret
metadata:
  name: harbor-credentials
  namespace: development
type: kubernetes.io/dockerconfigjson
data:
  # Base64-kodierter Docker-Config-String
  # Generieren mit: echo -n '{"auths":{"HARBOR-URL":{"username":"USER","password":"PASS
  .dockerconfigjson: <BASE64-ENCODED-STRING>
```

Füge zur .gitignore hinzu:

```
# Harbor Credentials - NIEMALS committen!
**/ImagePullSecret.yaml
**/harbor-credentials*.yaml
```

---

4. ArgoCD Integration

Was ist ArgoCD?

ArgoCD ist ein "Autopilot" für deine Deployments. Er: 1. Schaut in dein Git-Repository 2. Vergleicht: "Was steht in Git?" vs. "Was läuft im Cluster?" 3. Synchronisiert automatisch oder auf Knopfdruck

Klick-Anleitung für die ArgoCD-UI

Schritt 1: ArgoCD öffnen

1. Öffne deinen Browser
2. Gehe zu deiner ArgoCD-URL (z.B. `https://argocd.meinefirma.de`)
3. Logge dich ein mit deinen Zugangsdaten

Schritt 2: Neue Application erstellen

1. Klicke oben links auf **"+ NEW APP"** (blauer Button)

Schritt 3: General (Allgemeine Einstellungen)

Feld	Was eintragen	Erklärung
Application Name	`student-api`	Name deiner App in ArgoCD
Project	`default`	ArgoCD-Projekt (default ist ok für den Anfang)
Sync Policy	`Manual` oder `Automatic`	Manual = du klickst "Sync", Automatic = sofort bei Git-Änderung

Schritt 4: Source (Woher kommen die Dateien?)

Feld	Was eintragen	Erklärung
Repository URL	`https://github.com/DEIN-USER/manifest.git`	Dein Git-Repository
Revision	`HEAD` oder `main`	Welcher Branch
Path	`k8s/Student-api`	**WICHTIG!** Der Ordner mit deinen YAML-Dateien

Schritt 5: Destination (Wohin deployen?)

Feld	Was eintragen	Erklärung
Cluster URL	`https://kubernetes.default.svc`	Dein Kubernetes-Cluster (Standard für lokalen Cluster)
Namespace	`development`	In welchen Namespace deployen

Schritt 6: Erstellen

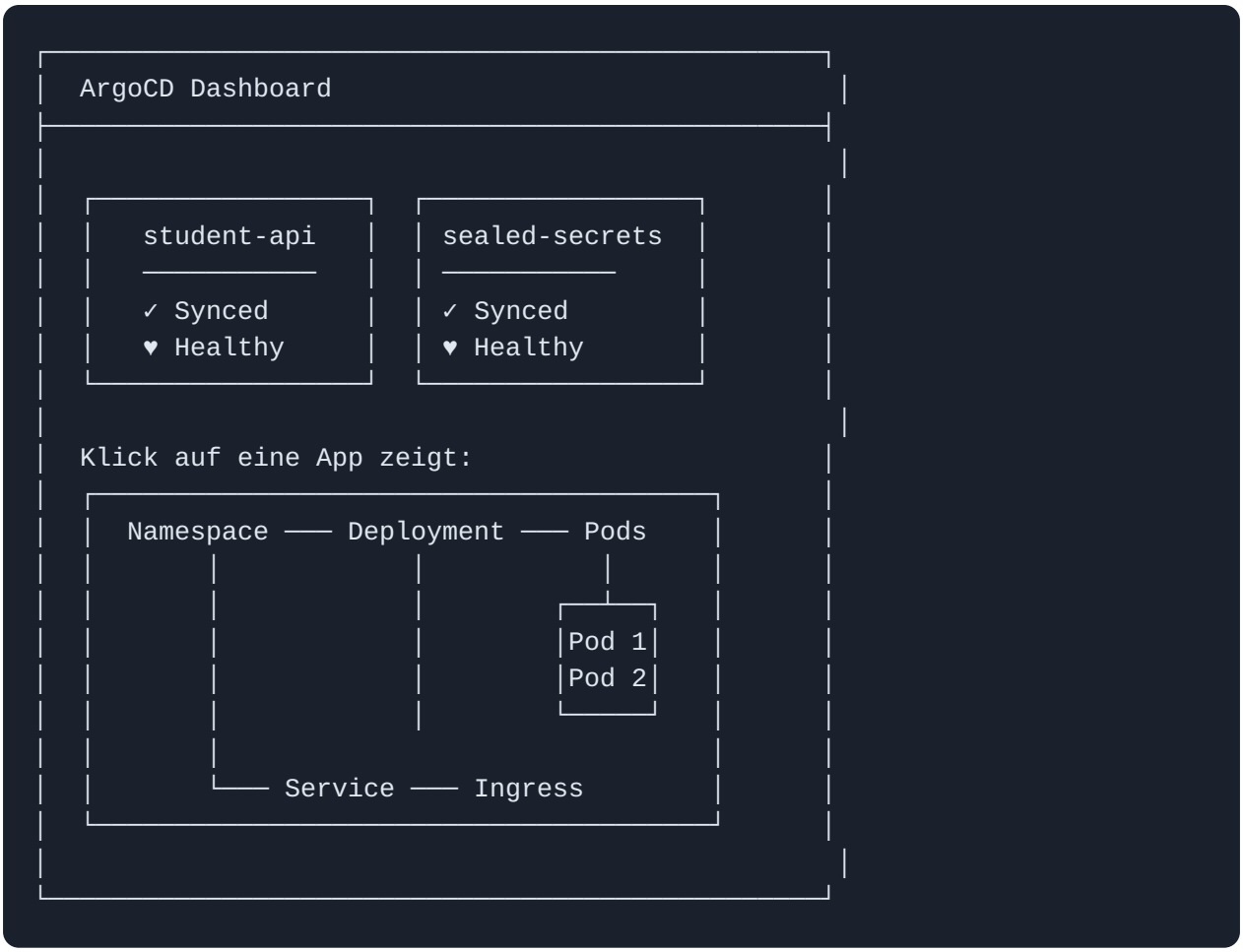
1. Scrolle nach unten
2. Klicke auf **"CREATE"** (blauer Button oben)

Schritt 7: Erste Synchronisierung

1. Du siehst jetzt deine App mit Status "OutOfSync" (gelb)

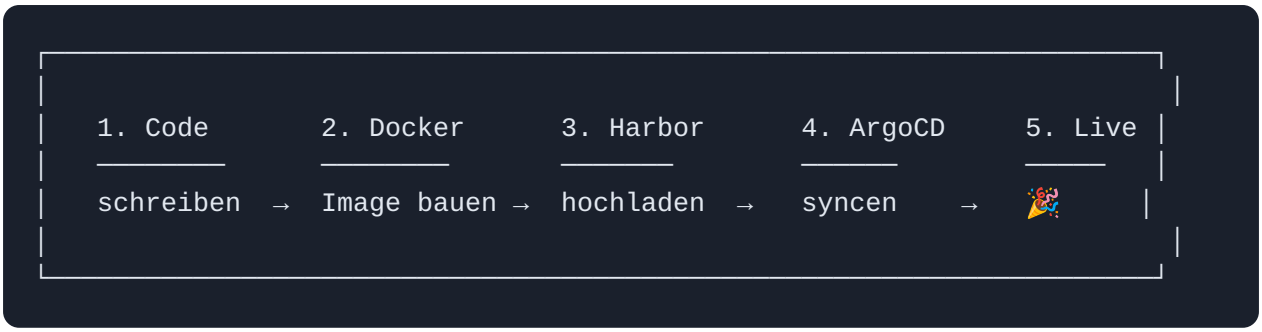
2. Klicke auf die App-Karte
3. Klicke oben auf **"SYNC"** (blauer Button)
4. Im Popup: Klicke **"SYNCHRONIZE"**
5. Warte bis alles grün ist = erfolgreich deployed!

Visuelle Darstellung in ArgoCD:



5. Deployment-Checkliste

Übersicht: Der komplette Workflow



Chronologische Checkliste

Phase 1: Vorbereitung (einmalig)

- [ ] **1.1** Docker installiert? `bash docker --version`
- [ ] **1.2** kubectl installiert und konfiguriert? `bash kubectl version kubectl cluster-info`
- [ ] **1.3** Zugang zu Harbor? (URL, Benutzername, Passwort notiert)
- [ ] **1.4** Zugang zu ArgoCD? (URL, Benutzername, Passwort notiert)

Phase 2: Namespace und Secrets erstellen (einmalig)

- [ ] **2.1** Namespace erstellen `bash kubectl apply -f k8s/Student-api/Namespace.yaml`
- [ ] **2.2** Harbor ImagePullSecret erstellen (siehe Kapitel 3) `bash kubectl create secret docker-registry harbor-credentials \ --namespace=development \ --docker-server=DEINE-HARBOR-URL \ --docker-username=DEIN-USER \ --docker-password=DEIN-PASSWORT \ --docker-email=DEINE-EMAIL`
- [ ] **2.3** Datenbank-Secret erstellen (oder Sealed Secret anwenden) `bash kubectl apply -f k8s/sealed-secrets/db-credentials-sealed.yaml`

Phase 3: Docker Image bauen und pushen

- [ ] **3.1** In Harbor einloggen `bash docker login harbor.meinefirma.de # Benutzername und Passwort eingeben`

- [ ] **3.2** Docker Image bauen `bash cd /pfad/zu/manifest docker build -t harbor.meinefirma.de/studenten/manifest-app:v1 ./StudentApi` **Erklärung:**
- `-t` = Tag/Name für das Image
- `./StudentApi` = Ordner mit dem Dockerfile
- [ ] **3.3** Image nach Harbor pushen `bash docker push harbor.meinefirma.de/studenten/manifest-app:v1`
- [ ] **3.4** In Harbor prüfen: Ist das Image angekommen?
- Harbor-UI öffnen → Projekt "studenten" → Image sollte sichtbar sein

Phase 4: Kubernetes-Manifeste anpassen

- [ ] **4.1** In `Deployment.yaml` die Image-URL anpassen: `yaml image: harbor.meinefirma.de/studenten/manifest-app:v1`
- [ ] **4.2** In `Ingress.yaml` den Host anpassen (falls nötig): `yaml host: student-api.deine-domain.de`
- [ ] **4.3** Änderungen committen und pushen: `bash git add . git commit -m "Update image and ingress configuration" git push`

Phase 5: Mit ArgoCD deployen

- [ ] **5.1** ArgoCD-UI öffnen
- [ ] **5.2** Falls noch nicht vorhanden: Application erstellen (siehe Kapitel 4)
- [ ] **5.3** Application synchronisieren:
- Auf App klicken → **"SYNC"** → **"SYNCHRONIZE"**
- [ ] **5.4** Warten bis alle Ressourcen grün sind (✓ Synced, ♥ Healthy)

Phase 6: Testen

- [ ] **6.1** Pods prüfen - laufen sie? `bash kubectl get pods -n development # STATUS sollte "Running" sein`
- [ ] **6.2** Logs prüfen - gibt es Fehler? `bash kubectl logs -n development -l app=student-api --tail=50`
- [ ] **6.3** Service prüfen: `bash kubectl get service -n development`
- [ ] **6.4** Ingress prüfen: `bash kubectl get ingress -n development`
- [ ] **6.5** Im Browser testen:

- Swagger UI: `http://student-api.deine-domain.de/swagger`
- API direkt: `http://student-api.deine-domain.de/api/student`

Phase 7: Bei Problemen - Debugging

- [ ] **7.1** Pod-Status prüfen: `bash kubectl describe pod -n development -l app=student-api`
- [ ] **7.2** Events im Namespace anzeigen: `bash kubectl get events -n development --sort-by='.lastTimestamp'`
- [ ] **7.3** Häufige Probleme:

Problem	Mögliche Ursache	Lösung
`ImagePullBackOff`	Image nicht gefunden oder keine Berechtigung	Harbor-URL und Secret prüfen
`CrashLoopBackOff`	App startet und stürzt ab	Logs prüfen (`kubectl logs`)
`Pending`	Nicht genug Ressourcen	Ressourcen-Requests reduzieren

Schnell-Referenz: Die wichtigsten Befehle

```
# === PODS ===
kubectl get pods -n development          # Alle Pods anzeigen
kubectl logs -n development <pod-name>   # Logs eines Pods
kubectl describe pod -n development <pod-name>  # Details eines Pods

# === DEPLOYMENTS ===
kubectl get deployments -n development   # Alle Deployments
kubectl rollout restart deployment/student-api -n development  # Neustart

# === SERVICES ===
kubectl get services -n development      # Alle Services
kubectl get ingress -n development       # Alle Ingress-Regeln

# === SECRETS ===
kubectl get secrets -n development       # Alle Secrets (Namen)
kubectl describe secret harbor-credentials -n development  # Secret-Details

# === DEBUGGING ===
kubectl get events -n development --sort-by='.lastTimestamp'  # Letzte Ereignisse
kubectl exec -it <pod-name> -n development -- /bin/sh  # In Pod "einloggen"

# === ARGOCD (CLI) ===
argocd app list                          # Alle ArgoCD-Apps
argocd app sync student-api              # App synchronisieren
```

Fazit

Du hast jetzt alles, was du brauchst:

1. ✅ **Ordnerstruktur** - Wo welche Dateien hingehören
2. ✅ **YAML-Dateien** - Vollständig kommentiert und erklärt
3. ✅ **Harbor-Zugang** - ImagePullSecret erstellen
4. ✅ **ArgoCD** - Schritt-für-Schritt UI-Anleitung
5. ✅ **Checkliste** - Vom Code bis zum Live-System

Tipp: Arbeite die Checkliste Schritt für Schritt durch. Bei Problemen: Erst Logs prüfen (`kubectl logs`), dann Events (`kubectl get events`).

Viel Erfolg beim Deployment! 🚀