# Konfiguration - WordPress

- **Verwendung**:
  WordPress wird als Container-Applikation im Kubernetes-Cluster betrieben. Die Konfiguration erfolgt über separate YAML-Dateien für Deployment, Netzwerk, Persistenz und Sicherheit.
- **Einsatzgrund**:
  WordPress ist ein etabliertes Content-Management-System (CMS) zur Erstellung und Verwaltung von Websites. Durch den Containerbetrieb werden Wiederverwendbarkeit, Skalierbarkeit und Portierbarkeit erhöht.
- **Rolle im System**:
  WordPress dient als zentrale Webplattform zur Veröffentlichung und Verwaltung von Webinhalten. Es wird über Ingress erreichbar gemacht und nutzt eine persistente MariaDB-Datenbank.

## Ressourcen - Anwendung

Im Folgenden sind alle YAML-Dateien aufgeführt, die zur Bereitstellung und Konfiguration der Anwendung benötigt werden. Sie decken u. a. Container-Deployment, Netzwerkzugriff, Speicheranbindung sowie Konfigurations- und Zugriffsdaten ab.

| Ressource | Dateiname | Zweck |
|---|---|---|
| Deployment | `deployment.yaml` | Startet den App-Container |
| Service | `service.yaml` | Interner Netzwerkzugriff |
| Persistent Volume Claim | `pvc.yaml` | Persistenz für Daten |
| ConfigMap | `configmap.yaml` | Konfigurationsparameter |
| Secret | `secret.yaml` | Zugangsdaten oder Tokens |
| Ingress | `wordpress-ingress.yaml` | Zugriff via Hostname |

## Ressourcen - Datenbank

Die folgenden YAML-Dateien definieren den Betrieb der zugehörigen Datenbank. Diese nutzt dieselben Konfigurationsdateien wie die Anwendung, um einheitliche und zentrale Umgebungsparameter sicherzustellen.

| Ressource | Dateiname | Zweck |
|---|---|---|
| Deployment | `db-deployment.yaml` | Startet die Datenbank (`MariaDB`) |
| Service | `db-service.yaml` | Intern erreichbar durch App |
| Persistent Volume Claim | `db-pvc.yaml` | Persistenz für Datenbankinhalte |

| Ressource | Dateiname | Zweck |
| --- | --- | --- |
| Konfiguration | - | Verwendet dieselbe ConfigMap & Secret wie die Anwendung |

## Deployment

> Definiert das Deployment für die Anwendung: Container-Image, Umgebungsvariablen, Volumes und Replikation.

deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wordpress-deployment
  namespace: m347-wordpress
  labels:
    app: wordpress
spec:
  replicas: 1
  selector:
    matchLabels:
      app: wordpress
  template:
    metadata:
      labels:
        app: wordpress
    spec:
      containers:
      - name: wordpress
        image: bitnami/wordpress:latest
        ports:
        - containerPort: 8080
        env:
        - name: WORDPRESS_DATABASE_HOST
          value: mariadb-service
        - name: WORDPRESS_DATABASE_USER
          valueFrom:
            secretKeyRef:
              name: secret
              key: username
        - name: WORDPRESS_DATABASE_PASSWORD
          valueFrom:
            secretKeyRef:
              name: secret
              key: password
        - name: WORDPRESS_DATABASE_NAME
          valueFrom:
            configMapKeyRef:
              name: configmap
              key: database_name
```

```
        - name: WORDPRESS_USERNAME
          valueFrom:
            secretKeyRef:
              name: secret
              key: WORDPRESS_USERNAME
        - name: WORDPRESS_PASSWORD
          valueFrom:
            secretKeyRef:
              name: secret
              key: WORDPRESS_PASSWORD
        volumeMounts:
        - mountPath: /bitnami/wordpress
          name: wordpress-persistent-storage
      volumes:
      - name: wordpress-persistent-storage
        persistentVolumeClaim:
          claimName: wordpress-pvc
```

### Erklärung der Konfiguration

- `replicas: 1`
  Es wird **nur ein Pod** instanziert, ausreichend für Entwicklungs- oder Testumgebungen.
- `image: bitnami/wordpress:latest`
  Verwendet ein vorgefertigtes, sicheres Image mit vorkonfiguriertem WordPress.
- `containerPort: 8080`
  Der Container ist auf Port `8080` erreichbar. Dies ist der Standard-Webport von WordPress.
- `env`
  Die Konfiguration erfolgt vollständig über Umgebungsvariablen, welche über `ConfigMap` und `Secret` eingebunden werden.
- `volumeMounts`
  Das WordPress-Dateiverzeichnis wird auf `/bitnami/wordpress` gemountet. Dort werden z. B. Uploads und Erweiterungen gespeichert.
- `persistentVolumeClaim`
  Das Deployment nutzt ein PVC (`wordpress-pvc`), welches in der separaten Datei `pvc.yaml` definiert ist, zur dauerhaften Speicherung von Daten.

## Service

> Stellt einen internen Kubernetes-Service zur Verfügung, über den die App im Cluster erreichbar ist.

`service.yaml`

```
apiVersion: v1
kind: Service
metadata:
```

```
  name: wordpress-service
  namespace: m347-wordpress
spec:
  selector:
    app: wordpress
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

### Erklärung der Konfiguration

- `kind: Service`
  Erstellt einen Kubernetes-internen Service, der WordPress im Cluster erreichbar macht.
- `selector.app: wordpress`
  Der Service wählt Pods aus, die mit dem Label `app: wordpress` gekennzeichnet sind.
- `ports`
  - `protocol: TCP`: Der Service kommuniziert via TCP-Protokoll.
  - `port: 80`: Port des Services innerhalb des Clusters.
  - `targetPort: 8080`: Port des Containers, auf den der Traffic weitergeleitet wird. Dies entspricht dem Webport des WordPress-Containers.

## Persistente Daten (PVC)

> Fordert persistenten Speicher im Cluster an, z. B. für Medien-Uploads oder Logs der Anwendung.

`pvc.yaml`

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: wordpress-pvc
  namespace: m347-wordpress
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: standard
```

### Erklärung der Konfiguration

- `kind: PersistentVolumeClaim`
  Fordert persistenten Speicherplatz an, um Daten dauerhaft im Cluster zu speichern.

- `accessModes: ReadWriteOnce`
  Der Speicher kann von genau einem Pod gleichzeitig im Schreibmodus verwendet werden.

- `resources.requests.storage: 10Gi`
  Die angeforderte Speichergrösse beträgt 10 Gibibyte. Hier werden beispielsweise Mediendateien, Themes oder Plugin-Daten gespeichert.

- `storageClassName: standard`
  Verwendet die standardmässig konfigurierte Storage-Class im Kubernetes-Cluster, welche bestimmt, wie und wo der Speicher bereitgestellt wird.

## Datenbank - Deployment

> Startet die zugehörige Datenbankinstanz inkl. Volume, Ports und Konfiguration.

`db-deployment.yaml`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mariadb-deployment
  namespace: m347-wordpress
  labels:
    app: mariadb
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mariadb
  template:
    metadata:
      labels:
        app: mariadb
    spec:
      containers:
      - name: mariadb
        image: mariadb:latest
        ports:
        - containerPort: 3306
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: secret
              key: root_password
        - name: MYSQL_DATABASE
          valueFrom:
            configMapKeyRef:
              name: configmap
              key: database_name
        - name: MYSQL_USER
          valueFrom:
```

```
            secretKeyRef:
              name: secret
              key: username
        - name: MYSQL_PASSWORD
          valueFrom:
            secretKeyRef:
              name: secret
              key: password
        volumeMounts:
        - mountPath: "/var/lib/mysql"
          name: mariadb-persistent-storage
      volumes:
      - name: mariadb-persistent-storage
        persistentVolumeClaim:
          claimName: mariadb-pvc
```

### Erklärung der Konfiguration

- `replicas: 1`
  Instanziert **einen Pod**, ausreichend für Entwicklungs- und Testzwecke.
- `image: mariadb:latest`
  Nutzt das offizielle Docker-Image von MariaDB in der aktuellsten Version.
- `ports`
  MariaDB läuft standardmässig auf Port `3306`, der hier für Datenbankzugriffe bereitgestellt wird.
- `env`
  MariaDB erhält Konfigurationsparameter über Secrets und eine ConfigMap:
  - `MYSQL_ROOT_PASSWORD`: Root-Passwort für die Datenbank (Secret).
  - `MYSQL_DATABASE`: Name der initialen Datenbank (ConfigMap).
  - `MYSQL_USER`: Benutzername für die WordPress-Datenbank (Secret).
  - `MYSQL_PASSWORD`: Passwort für den Datenbankbenutzer (Secret).
- `volumeMounts`
  Persistenter Speicher ist auf dem Containerpfad `/var/lib/mysql` eingebunden. So bleiben Datenbankinhalte bei Neustarts erhalten.
- `persistentVolumeClaim`
  Das Deployment verwendet ein PVC (`mariadb-pvc`), definiert in der separaten Datei `db-pvc.yaml`, zur dauerhaften Speicherung der Datenbankinhalte.

## Datenbank - Service

> Stellt einen internen Kubernetes-Service für die Datenbank bereit, der durch die App genutzt wird.

`db-service.yaml`

```
apiVersion: v1
kind: Service
metadata:
  name: mariadb-service
  namespace: m347-wordpress
spec:
  selector:
    app: mariadb
  ports:
  - protocol: TCP
    port: 3306
    targetPort: 3306
```

### Erklärung der Konfiguration

- `kind: Service`
  Erstellt einen Kubernetes-internen Service, der MariaDB im Cluster bereitstellt.
- `selector.app: mariadb`
  Der Service wählt die Pods mit dem Label `app: mariadb` aus, um Anfragen an die richtigen Pods weiterzuleiten.
- `ports`
  - `protocol: TCP`: Nutzt das TCP-Protokoll für die Kommunikation.
  - `port: 3306`: Der Port, über den der Service innerhalb des Clusters erreichbar ist.
  - `targetPort: 3306`: Der Port des MariaDB-Containers, auf den die Anfragen weitergeleitet werden.

## Datenbank - Persistente Daten (PVC)

> Bindet ein Volume für die dauerhafte Speicherung von Datenbankdaten ein.

`db-pvc.yaml`

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mariadb-pvc
  namespace: m347-wordpress
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: standard
```

### Erklärung der Konfiguration

- `kind: PersistentVolumeClaim`
  Fordert persistenten Speicherplatz zur dauerhaften Speicherung der MariaDB-Daten an.
- `accessModes: ReadWriteOnce`
  Der Speicher wird von genau einem Pod gleichzeitig im Schreibmodus genutzt.
- `resources.requests.storage: 10Gi`
  Die Grösse des angeforderten Speicherplatzes beträgt 10 Gibibyte, ausreichend für Entwicklungs- und Testdatenbanken.
- `storageClassName: standard`
  Nutzt die Standard-StorageClass des Kubernetes-Clusters, welche definiert, wie und wo der Speicher bereitgestellt wird.

### ConfigMap & Secret

> Definiert zentrale Konfigurationswerte (ConfigMap) und vertrauliche Daten (Secret), die in App und DB referenziert werden.

## ConfigMap - `configmap.yaml`

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: configmap
  namespace: m347-wordpress
data:
  database_name: wordpress
```

### ConfigMap - Erklärung der Konfiguration

Die `ConfigMap` enthält zentrale, **nicht-sensitive Konfigurationswerte** für die WordPress-Instanz. Diese Werte werden in den Deployments der App- und Datenbank-Container als Umgebungsvariablen eingebunden.

**Begründung:**

- Durch die Auslagerung in eine `ConfigMap` können Konfigurationswerte unabhängig vom Containerimage definiert und bei Bedarf zentral angepasst werden.
- Sensible Daten (wie Passwörter) werden **nicht** hier, sondern im `Secret` verwaltet, was die Sicherheit erhöht.

## Secret - `secret.yaml`

```
apiVersion: v1
kind: Secret
metadata:
```

```
    name: secret
    namespace: m347-wordpress
  type: Opaque
  stringData:
    username: d29yZHByZXNz # wordpress
    password: cGFzc3dvcmQ= # password
    root_password: cm9vdF9wYXNzd29yZA== # root_password
    WORDPRESS_PASSWORD:  einSicheresPasswort123
    WORDPRESS_USERNAME: admin
```

### Secret - Erklärung der Konfiguration

Die `Secret` enthält **sensible Zugangsdaten**, die für den Betrieb der WordPress-Instanz notwendig sind - insbesondere zur Authentifizierung bei der MariaDB-Datenbank. Sie wird im Deployment von App- und Datenbank-Container referenziert.

**Begründung:**

- Secrets werden verwendet, um vertrauliche Informationen wie Passwörter, Tokens oder API-Keys **verschlüsselt im Cluster** zu speichern.
- Die Werte unter `stringData` können **leserlich im YAML** definiert werden - Kubernetes wandelt sie beim Speichern automatisch in Base64-kodierte Einträge um.
- Der Typ `Opaque` ist der Standardtyp für frei definierte Schlüssel-Wert-Paare.

> [!NOTE] **Best Practices im produktiven Betrieb**:
>
> - Secrets sollten **nicht versioniert** oder öffentlich gespeichert werden.
> - Der Zugriff auf Secrets sollte im Cluster über **Rollen & Rechte** (RBAC) eingeschränkt werden.

### Ingress / Externer Zugriff

> Regelt den externen Zugriff auf die Anwendung über Hostnamen mithilfe eines Ingress Controllers.

Die Datei `wordpress-ingress.yaml` definiert, unter welchem Hostnamen (`wordpress.m347.ch`) die WordPress-Applikation von ausserhalb des Clusters erreichbar ist.
Sie verweist auf den zentralen Ingress Controller und sorgt für die Weiterleitung eingehender Anfragen an den zugehörigen Service der WordPress-Anwendung.
Da das zugrundeliegende Ingress-System für alle Anwendungen identisch ist, wird die übergeordnete Konfiguration des Ingress Controllers inklusive Routingprinzipien und Klassendefinition zentral in der Konfigurationsdatei des Ingress Controllers dokumentiert.

## Besonderheiten & Herausforderungen

Bei der Umsetzung und Bereitstellung der WordPress-Anwendung im Kubernetes-Cluster zeigte sich deutlich, wie stark standardisiert und gleichzeitig eingeschränkt man in der Gestaltung der YAML-Konfigurationsdateien tatsächlich ist. Kubernetes gibt eine strikte Struktur und klare Anforderungen für Ressourcen wie Deployments, Services, PersistentVolumeClaims und Secrets vor, die zwingend eingehalten werden müssen. Dadurch bleibt wenig Raum für individuelle Anpassungen oder kreative Lösungen. Dies mag auf den ersten Blick einschränkend wirken, ist jedoch einer der zentralen Vorteile von Kubernetes: Die Standardisierung sorgt für

einheitliche, zuverlässige und reproduzierbare Abläufe beim Bereitstellen und Verwalten von Anwendungen. Die Herausforderung bestand weniger darin, eigene kreative Lösungen zu entwickeln, als vielmehr darin, die bestehenden Vorgaben präzise umzusetzen. Es war überraschend und zugleich lehrreich, festzustellen, wie stark Kubernetes die konkrete Umsetzung beeinflusst und gleichzeitig vereinfacht – vorausgesetzt, man hält sich genau an die vorgegebenen Standards.