

Tag 3: GitOps, Docker in der Entwicklung und Deployment-Strategien

19.06.2024, Daniel Krämer & Malte Fischer

© Copyright 2024 anderScore GmbH

- **Tag 1 – Einführung in Git und GitLab, Git-Workflow im Team**
 - Einführung & Kursüberblick
 - Grundlagen von Git
 - Git Rebase und Merge-Strategien
 - Git Remote
 - Grundlagen von GitLab
 - Git-Workflow im Team
- **Tag 2 – Vertiefung Git-Workflow, CI/CD & GitLab CI**
 - Gitflow-Workflow
 - Tags, Releases & deren Verwaltung
 - GitLab-Runner
 - Einführung in GitLab CI/CD & gitlab.yml
- **Tag 3 – GitOps, Docker in der Entwicklung und Deployment-Strategien**
 - GitOps Grundlagen
 - Lokale Entwicklung mit Docker
 - Container/Docker-Registry
 - Erstellen von Release- und Tagged-Images
 - Möglichkeiten des Deployments & Verwaltung von Konfiguration
 - Abschlussübung & Diskussion

- **Tag 1 – Einführung in Git und GitLab, Git-Workflow im Team**
 - Einführung & Kursüberblick
 - Grundlagen von Git
 - Git Rebase und Merge-Strategien
 - Git Remote
 - Grundlagen von GitLab
 - Git-Workflow im Team
- **Tag 2 – Vertiefung Git-Workflow, CI/CD & GitLab CI**
 - Gitflow-Workflow
 - Tags, Releases & deren Verwaltung
 - GitLab-Runner
 - Einführung in GitLab CI/CD & gitlab.yml
- **Tag 3 – GitOps, Docker in der Entwicklung und Deployment-Strategien**
 - GitOps Grundlagen
 - Lokale Entwicklung mit Docker
 - Container/Docker-Registry
 - Erstellen von Release- und Tagged-Images
 - Möglichkeiten des Deployments & Verwaltung von Konfiguration
 - Abschlussübung & Diskussion

Erstellen von

Release- und Tagged-Images

Inhalt

- Tagging von Docker Images
- Strategien zum Image Tagging
- Verwendung mit GitLab

Tagging von Docker Images

- Was ist Tagging?
- Warum Tagging?
- Tagging während des Builds
- Tagging nach dem Build
- Best Practices

Was ist Tagging?

- Docker Image: unique ID
 - Beispiel: myimage:1f6ad45c7b3
 - Arbeiten mit IDs umständlich
- Lesbare Alternative... Image Tagging!
- Vergleichbar mit Labeling (Beschriftung)
- Erlauben aussagekräftige Namen
 - Leichter zu identifizieren
 - Einfacher zu benutzen
- Image Name = Repository Name
- Tag = optionaler Identifier
- Beispiel: ubuntu:24.04

Warum Tagging?

- Lesbarkeit
 - ID vs. Tag (lesbar und benutzerfreundlicher)
- Versionskontrolle
 - Wartung verschiedener Versionen
- Rückverfolgbarkeit und Verantwortlichkeit
 - Herkunft und Verlauf eines Builds
 - Historie
- Convenience
 - Leicht verwendbar
- Vereinfachtes Deployment & Automatisierung
 - Durch konsistente Tagging-Strategie

Tagging während des Builds

- Mit dem `-t` Flag während des Build-Prozesses

```
docker build -t [repository]:[TAG] .
```

Tagging nach dem Build

- Vorhandene Images mit `docker tag` Befehl taggen

```
docker tag [IMAGE_ID] [repository]:[TAG]
```

Best Practices

- Aussagekräftige Tags
 - Selbstbeschreibend
 - Version oder Zustand
- Konsistenz
 - Einheitliches Tagging-Schema
- Regelmäßige Updates
 - Tags immer aktualisieren
 - → Strategien zum Image Tagging

Strategien zum Image Tagging

- Image ID (digest)
- Image Tags:
 - Rolling Tags
 - Git Tags
 - Branch Names
 - SemVer Tags (Semantic Versioning)
 - Git Commit Hash
 - Timestamp / Date-based Tags
 - Build ID

Rolling Tags

- Weit verbreitet → :latest und :stable
- Relevanteste und neuste Versionen
- Vorsicht: Volatiler Inhalt!
 - Für Test-Stage OK, bei Produktion No-Go
 - Bei Produktion besser: Unique Tags
- Rollback schwierig
- Herausforderung
 - Image IDs (digest): schwer lesbar
 - Image Tags: veränderlich

Git Tags

- Nützlich bei Verwendung von Git Tags für Releases
- Konsistenz zwischen Versionsverwaltung und Deployable
- Git Tag „v2.5.1“
 - → Gleichen Tag als Docker Image Tag

Branch Names

- Branching Strategie
 - Branch-Namen verwenden, um Tags zu managen
- Beispiel
 - Branch: release/2.5.1 für ein spezifisches Release
 - Docker Image mit 2.5.1 taggen

SemVer Tags (Semantic Versioning)

- Anstatt zufällige Namen spezifische Versionsnummer
- „Spezialfall“ des :stable Tag (Grundidee)
- Notation MAJOR.MINOR.PATCH
 - Beispiel: 2.5.1
 - MAJOR = Inkompatible Änderungen
 - MINOR = Kompatible Änderungen
 - PATCH = Patches
- Neuer Build mit kleinsten Änderungen = Patchnummer hochzählen
 - → aus 2.5.1 wird 2.5.2
- Tags weiterhin mutable

Git Commit Hash

- Neuer Commit = neues Image
- Git Hash zum Tagging
 - Kürzer als Docker Image Digests
- Traceability (Rückverfolgbarkeit) sehr hoch!
- Nicht selbsterklärend
- Beispiel
 - 2.5.1-sha1abcde

Timestamp / Date-based Tags

- Unique Identifier
 - → „Semi-immutable“ Referenz
- Automatisch generiert → einzigartig
- Einfache Lösung mit vielen Nachteilen
 - Release am 20.05.2024, Tagging → 2.5.1-20240520
 - Zeitzonen sind böse!
 - Korrelation zum Changeset fehlt

Build ID

- Unique Identifier
 - → „Semi-immutable“ Referenz
- Automatisch generiert → einzigartig
- Referenziert bestimmten Build
- Kann (theoretisch) nicht gefaked werden
- Analog zum Image Digest
 - → Keine Hinweise auf Änderungen vom Release
 - Auch nicht hilfreich beim Suchen nach einem bestimmten Image

Use Cases für die Strategien

- Rolling Tags
 - Base Images, welche immer aktuell sein sollen
- Unique Tags
 - Software in Produktion
 - Empfehlung: Build ID Tag
- SemVer
 - Koppelt ein Image ans darunterliegende Changeset
 - Kann automatisiert werden
 - Nutzer erhalten kompatiblns Build für ihre Anwendungen
- Kombination möglich!

Verwendung mit GitLab

- Authentifizierung an der Container Registry
- Authentifizierung in CI/CD Pipelines
- Images bauen und pushen
- Container Registry: Beispiele mit GitLab CI/CD

Authentifizierung an der Container Registry

- Unterstützte Mechanismen
 - Personal Access Token
 - Deploy Token
 - Project Access Token
 - Group Access Token
- Erforderliche Berechtigungen (Scopes)
 - *read_registry* für read (pull)
 - *write_registry* und *read_registry* für write (push)

Authentifizierung an der Container Registry

```
docker login registry.example.com
```

oder

```
TOKEN=<token>
```

```
echo "$TOKEN" | docker login registry.example.com -u  
<username> --password-stdin
```

Authentifizierung in CI/CD Pipelines

- Variable CI_REGISTRY_USER
 - Benutzer/Job mit read + write Scope
 - Passwort automatisch: CI_REGISTRY_PASSWORD
 - `echo "$CI_REGISTRY_PASSWORD" | docker login $CI_REGISTRY -u $CI_REGISTRY_USER --password-stdin`
- CI Job Token
 - `echo "$CI_JOB_TOKEN" | docker login $CI_REGISTRY -u $CI_REGISTRY_USER --password-stdin`

Authentifizierung in CI/CD Pipelines

- read (pull) access → read_registry
- write (push) access → read_registry & write_registry
- Deploy Token
 - `echo "$CI_DEPLOY_PASSWORD" | docker login $CI_REGISTRY -u $CI_DEPLOY_USER --password-stdin`
- Personal Access Token
 - `echo "<access_token>" | docker login $CI_REGISTRY -u <username> --password-stdin`

Images bauen und pushen

1. An der Container Registry authentifizieren
 2. Docker CLI nutzen
 1. Build:
`docker build -t registry.example.com/group/project/image .`
 2. Push:
`docker push registry.example.com/group/project/image`
- CI/CD fürs Testen, Bauen, Pushen und Deployen

Docker-in-Docker (dind)

- Unterstützte Executors
 - Docker Executor
 - Kubernetes Executor
- Executor verwendet *docker* Container Image
 - Image beinhaltet alle docker tools
 - Job-Script im privilegierten Modus
- Spezifische Version nutzen!
 - Beispiel: `docker:24.0.5`
 - Ansonsten bei `:latest` Inkompatibilitätsprobleme

.gitlab-ci.yml

- Ermöglicht Bauen und Pushen von Images in die Registry
- Mehrere Jobs authentifizieren
 - `before_script`
- `docker build --pull`
 - Änderungen am Base Image
 - Pro: Base Image ist up-to-date
 - Contra: Build dauert länger
- Vor `docker run` ein `docker fetch`
 - Fetched aktuelles Image
 - Wichtig bei mehreren Runnern, welche Images lokal cachen

Beispiel: Docker-in-Docker Container Image (**Container Registry**)

Eigene Container Images mit Docker-in-Docker bauen

1. Docker-in-Docker einrichten
2. **image** und **service** auf *docker* bzw. *docker-dind* Image zeigen lassen
3. **alias** für *docker-dind* definieren

`.gitlab-ci.yml`

`build:`

`image: $CI_REGISTRY/group/project/docker:20.10.16`

`services:`

`- name: $CI_REGISTRY/group/project/docker:20.10.16-dind`

`alias: docker`

`stage: build`

`script:`

`- docker build -t my-docker-image .`

`- docker run my-docker-image /script/to/run/tests`

Beispiel: Docker-in-Docker Container Image (Container Registry)

.gitlab-ci.yml

build:

image: \$CI_REGISTRY/group/project/docker:20.10.16

services:

- name: \$CI_REGISTRY/group/project/docker:20.10.16-dind

alias: docker

stage: build

script:

- docker build -t my-docker-image .
- docker run my-docker-image /script/to/run/tests

- Ohne **alias** kann das *docker* Container Image keinen Docker Host finden und folgende Fehlermeldung erscheint:
 - error during connect: Get http://docker:2376/v1.39/info: dial tcp: lookup docker on 192.168.0.1:53: no such host

Dependency Proxy

- Lokaler Proxy
 - Für häufig genutzte Upstream-Images
 - pull through cache für DockerHub
 - Sicht des Docker Clients: Weitere Registry
 - Verbessert Performance bei häufigen Builds

Dependency Proxy

- Docker Hub Rate Limiting
 - <https://docs.docker.com/docker-hub/download-rate-limit/>
 - Begrenzt die Image Pulls
 - Pro Commit eine Pipeline angestoßen
 - Selbst bei gleichem Image → Docker Pull Count erhöht durch „manifest requests“
 - Manifest („Inhaltverzeichnis des Images“)
 - Informationen über Layers und Blobs des Images
- Dependency Proxy GitLab Dokumentation:
https://docs.gitlab.com/ee/user/packages/dependency_proxy/
- Hier: Keine weitere Verwendung!

Aufgabe 1: Einfache Docker-in-Docker Build-Pipeline

1. Ziel: Verständnis von dind schaffen

2. Schritte:

- .gitlab-ci.yml dem Projekt hinzufügen oder vorhandene nutzen
- Als image folgendes verwenden: `docker:20.10.16`
- Die stage sollte `build` sein
- Als service das Image als `-dind` einbinden
- Im script Teil sollte folgendes passieren
 1. Bei der Container Registry einloggen (`docker login`)
 2. Das Container Image aus dem aktuellen Projekt bauen (`docker build`)
 3. Das gebaute Image in die Registry pushen (`docker push`)

Aufgabe 2: Docker-in-Docker mit Variablen erweitern

1. Ziel: Verständnis der Variablen schärfen

2. Schritte:

- Fügen Sie die Variable `IMAGE_TAG` hinzu
- Nutzen Sie die neue Variable im script Teil

3. Hinweise:

- `IMAGE_TAG` wird später beim build und push benötigt
- Beim Docker-in-Docker Container Image haben Sie GitLab-predefined-Variables kennengelernt
- `$CI_COMMIT_REF_SLUG` ist eine vordefinierte Variable in GitLab und ist der Branch- oder Tag-Name sanitized und lowercase

Lösung 1: Simples Docker-in-Docker

`.gitlab-ci.yml:`

`build:`

`image: docker:20.10.16`

`stage: build`

`services:`

- `- docker:20.10.16-dind`
`alias: docker`

`script:`

- `- docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY`
- `- docker build -t $CI_REGISTRY/group/project/image:latest .`
- `- docker push $CI_REGISTRY/group/project/image:latest`

Lösung 2: Docker-in-Docker mit Variablen

`.gitlab-ci.yml:`

`build:`

`image: docker:20.10.16`

`stage: build`

`services:`

`- docker:20.10.16-dind`

`variables:`

`IMAGE_TAG: $CI_REGISTRY_IMAGE:$CI_COMMIT_REF_SLUG`

`script:`

`- docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY`

`- docker build -t $IMAGE_TAG .`

`- docker push $IMAGE_TAG`

- `$CI_REGISTRY_IMAGE`
 - Ist die Adresse der Registry des aktuellen Projektes
- `$CI_COMMIT_REF_NAME`
 - Ist der Branch- oder Tag-Name
 - In lowercase und sanitized als Variable `$CI_COMMIT_REF_SLUG`