dataport

anderScore
trust in competence

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

19.06.2024, Daniel Krämer & Malte Fischer

Agenda

- **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

Agenda

- **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

Release- und Tagged-Images

anderScore trust in competence

Inhalt

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

Release- und Tagged-Images

anderScore trust in competence

Tagging von Docker Images

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

Release- und Tagged-Images

anderScore trust in competence

Was ist Tagging?

- Jedes Docker Image hat eine unique ID
  - Mit IDs zu arbeiten kann umständlich sein
- Lesbare Alternative… Image Tagging!
- Tagging vergleichbar mit Labeling (Beschriftung)
- Tags erlauben aussagekräftige Namen
  - Leichter zu identifizieren
  - Einfacher zu benutzen

- Image Name = Repository Name
- Tag = optionaler Identifier
- Beispiel: `Ubuntu:24.04`

Release- und Tagged-Images

anderScore trust in competence

Warum Tagging?

- Lesbarkeit
  - Im Vergleich zu IDs besser lesbar und benutzerfreundlicher
- Versionskontrolle
  - Helfen beim Maintaining von verschiedenen Versionen des Images
- Rückverfolgbarkeit und Verantwortlichkeit
  - Herkunft und Verlauf eines Builds
  - Wann wurde was durch wen aktualisiert/geändert
- Convenience
  - In Befehlen und Skripten leichter zu verwenden
- Vereinfachtes Deployment & Automatisierung
  - Konsistente Tagging-Strategie macht automatisierte Deployments einfacher
    - CI/CD Pipeline automatisch: build, test and deploy ☺

Release- und Tagged-Images

anderScore trust in competence

Tagging während des Builds

- Mit dem –t Flag während des Build-Prozesses

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

Release- und Tagged-Images

anderScore trust in competence

Tagging nach dem Build

- Vorhandene Images mit `docker tag` Befehl taggen

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

Release- und Tagged-Images

anderScore trust in competence

Best Practises

- Aussagekräftige Tags
  - Tags sollten beschreibend (deskriptive) sein
  - Die Image Version oder Zustand wiedergeben
- Konsistenz
  - Einheitliches Tagging-Schema für verschiedene Images und Versionen
- Regelmäßige Updates
  - Tags immer aktualisieren, gerade bei einer neuen Version
  - → Strategien zum Image Tagging

Release- und Tagged-Images

anderScore trust in competence

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

Release- und Tagged-Images

anderScore trust in competence

Rolling tags

- Zwei weit verbreitete→`:latest` und `:stable`
- Relevanteste und neuste Build-Tag
- Vorsicht: Inkompatibles Image!
  - Für Test-Stage OK, Production No-Go
  - Production besser: unique Tags
- Schwierig zu einer früheren Version zurückzukehren
- Herausforderung
  - Image IDs (digest): not human-readable
  - Image tags: mutable

Release- und Tagged-Images

anderScore trust in competence

Git Tags

- Nützlich, wenn man bereits Git Tags für Releases nutzt
- Diese Tags können direkt als Docker Image Tags genutzt werden
- Git Tag „v2.5.1“
  - → Gleichen Tag als Docker Image Tag verwenden

Release- und Tagged-Images

anderScore trust in competence

Branch Names

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

Release- und Tagged-Images

anderScore trust in competence

SemVer tags (Semantic Versioning)

- Anstatt zufällige Namen direkt Nummerierung
- „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

Release- und Tagged-Images

anderScore trust in competence

Git Commit Hash

- Mit jedem Commit ein neues Docker Image
- Kurzen Git Hash zum Tagging nutzen
  - Sind kürzer als Image Digests
- Traceability (Rückverfolgbarkeit) sehr hoch!
- Tags allerdings nicht selbsterklärend
- Beispiel
  - 2.5.1-sha1abcde

Release- und Tagged-Images

anderScore trust in competence

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`
  - Timezonen sind böse!
  - Korrelation zum enthaltenden Changeset fehlt
  - Image mit demselben Tag manuell pushen

Release- und Tagged-Images

anderScore trust in competence

Build ID

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

Release- und Tagged-Images

anderScore trust in competence

Use Cases für die Strategien

- Rolling tags
  - Für Base Images, welche immer aktuell sein sollen
- Unique tags
  - Wenn Container in die Production gehen
  - Empfehlung: Build ID Tag
- SemVer
  - Koppelt ein Image ans darunterliegende Changeset
  - Kann automatisiert werden
  - Nutzer kriegen kompatibles Build für ihre Anwendungen
- Rolling und SemVer lassen sich gut kombinieren
- In kleinen Teams mit manuell überschaubaren Umfang
  - Digests, Git Commit Hash, Timestamps oder Build IDs nutzbar

Release- und Tagged-Images

anderScore trust in competence

Verwendung mit GitLab

- Authentifizierung mit der Container Registry
- GitLab CI/CD zum authentifizieren
- Images bauen und pushen
  - Docker
  - GitLab CI/CD
    - Docker-in-Docker Container Image (Container Registry)
    - Docker-in-Docker Container Image (Dependency Proxy)
- Container Registry Beispiele mit GitLab CI/CD

Release- und Tagged-Images

anderScore trust in competence

Authentifizierung mit der Container Registry

- Verschiedene Möglichkeiten
  - Personal access token
  - Deploy token
  - Project access token
  - Group access token
- Alle Methoden erfordern einen Mindestumfang:
  - Für read (pull) in der read_registry
  - Für write (push) in der write_registry und read_registry
- Zum Authentifizieren
  - `docker login registry.example.com`
  - oder
  - `TOKEN=<token>`
  - `echo "$TOKEN" | docker login registry.example.com -u <username> --password-stdin`

Release- und Tagged-Images

anderScore trust in competence

CI/CD zur Authentifizierung bei der Container Registry

- CI/CD Variable: CI_REGISTRY_USER
  - Job-bezogener Benutzer mit Lese- und Schreibrechten in der CR
  - Passwort automatisch erzeugt: 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 $CI_REGISTRY_USER --password-stdin`
- Für read (pull) access → read_registry
- Für write (push) access → read_registry & write_registry
  - Deploy Token
    - `echo "$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`

Release- und Tagged-Images

anderScore trust in competence

Images bauen und pushen

1. Mit der Container Registry authentifizieren
2. Docker 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

Release- und Tagged-Images

anderScore trust in competence

Docker-in-Docker (`dind`)

- Registrierter Runner nutzen dind automatisch
  - Docker Executor oder
  - Kubernetes Executor
- Executor nutzt ein Container Image von Docker
  - Bereitgestellt von Docker, um die CI/CD jobs auszuführen
- Docker Image beinhaltet alle docker tools
  - Und kann das Job-`script` im Kontext des Images im privilegierten Modus ausführen
- Immer eine spezifische Version nutzen!
  - Beispiel: `docker:24.0.5`
  - Ansonsten bei `:latest` Inkompatibilitätsproblemen, falls Update des Images

Release- und Tagged-Images

anderScore trust in competence

.gitlab-ci.yml

- Bauen und Pushen von Images in die Registry
- Falls mehrere jobs Authentifizierung benötigen
  - Befehl zum Authentifizieren im `before_script`
- `docker build –pull` um Änderungen am Base Image zu ziehen
  - Build dauert dadurch länger, aber das Image ist up-to-date
- Vor jedem `docker run` ein `docker fetch`
  - Um das aktuelle Image zu fetchen
  - Besonders wichtig bei mehreren Runnern, welche Images lokal cachen

Release- und Tagged-Images

anderScore trust in competence

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

Eigene Container Images mit Docker-in-Docker nutzen

1. Docker-in-Docker einrichten
2. `image` und `service` auf die Registry zeigen lassen
3. `alias` hinzufügen für den `service`

`.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
```

Release- und Tagged-Images

anderScore trust in competence

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 service alias kann das Container Image den `dind` service nicht 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`

Release- und Tagged-Images

anderScore trust in competence

Beispiel: Docker-in-Docker Container Image (Dependency Proxy)

Eigene Container Images mit Docker-in-Docker nutzen

1. Docker-in-Docker einrichten
2. `image` und `service` auf die Registry zeigen lassen
3. `alias` hinzufügen für den `service`

`.gitlab-ci.yml`

```
build:
  image: ${CI_DEPENDENCY_PROXY_GROUP_IMAGE_PREFIX}/docker:20.10.16
  services:
    - name: ${CI_DEPENDENCY_PROXY_GROUP_IMAGE_PREFIX}/docker:18.09.7-dind
      alias: docker
  stage: build
  script:
    - docker build -t my-docker-image .
    - docker run my-docker-image /script/to/run/tests
```

Release- und Tagged-Images

Docker-in-Docker Container Image (Dependency Proxy)

.gitlab-ci.yml

```
build:
  image: ${CI_DEPENDENCY_PROXY_GROUP_IMAGE_PREFIX}/docker:20.10.16
  services:
    - name: ${CI_DEPENDENCY_PROXY_GROUP_IMAGE_PREFIX}/docker:18.09.7-dind
      alias: docker
  stage: build
  script:
    - docker build -t my-docker-image .
    - docker run my-docker-image /script/to/run/tests
```

- Ohne service alias kann das Container Image den `dind` service nicht 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`

Release- und Tagged-Images

anderScore trust in competence

Dependency Proxy

- Lokaler Proxy
  - Genutzt für häufig genutzte Upstream-Images
  - Agiert als pull through cache für DockerHub
  - Aus Sicht des Docker Clients: Einfach eine weitere Registry
- Docker Hub rate limiting
  - https://docs.docker.com/docker-hub/download-rate-limit/
  - Begrenzt die Image pulls von Docker Hub
  - Meist läuft bei jedem commit eine Pipeline
    - Selbst bei gleichem Image → Docker Pull Count erhöht durch „manifest requets“
  - 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!

Release- und Tagged-Images

anderScore trust in competence

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

1. **Ziel:** Verständis 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` verwenden
   - Im `script` Teil sollte folgendes passieren
     1. Bei der Container Registry einloggen (docker login)
     2. Das Container Image aus dem aktueleln Projekt bauen (docker build)
     3. Das gebaute Image in die Registry pushen (docker push)

Release- und Tagged-Images

anderScore trust in competence

Aufgabe 2: Docker-in-Docker mit Variablen erweitern

1. **Ziel:** Verständis 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

Release- und Tagged-Images

anderScore trust in competence

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
```

Release- und Tagged-Images

anderScore trust in competence

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 Projekts
- $CI_COMMIT_REF_NAME
  - Ist der Branch- oder Tag-Name
  - In sowercase und sanitized als Variable $CI_COMMIT_REF_SLUG