Problems (*and Solutions*) with Docker Images in MSD

Mira Hedl
Oct / 31 / 2017 (~= Dec / 25 / 1039)

About me

Mira Hedl <dingo@matfyz.cz> - DevOps engineer / system architect

Years of experience:

- SW Development **for fun** (20 years) [Pascal, Assembler, C, Perl, Lua]
- SW Development **for living** (9 years) [C, C++, C#, Groovy, Python, Scheme, Lua, Yang]
- Networking **for living** (6 years) [Cisco, Juniper, CUNI netadmin, router OS developer]
- QA / Testing **for living** (1 year) [Jenkins, nUnit/jUnit/nose, Unit and Functional tests]
- Operations **for living** (5 years)
- Dev support **for living** (3 years, started in MSD)
- **Docker for fun** (4 years, since Aug/2013, docker v0.6)
- **Docker in production** (0-2 years, depends on perspective)

Docker creator, **Solomon Hykes** (and me).

June 2016
Seattle, WA

Agenda

- Short and brief history - overview and patterns that emerged across multiple users
- Docker system architecture decisions -- discussing PROS and CONS
- Overall mess and disorder resulting out of these defaults
- Work-arounds, limitations, tips and good practices to mitigate that CONS
- Current solution for End-to-end pipeline for making Docker Images in MSD

History of **PoC** docker image registries

Our story with docker images (from admin POV): [should provide **context for the talk**]

1. Hurray, docker registry (1. **registry:2**, 2. **DTR 1.6**, 3. **DTR 2.1**, 4. **portus**)
2. Sobering…
   - Problems with image **ownership**
   - Problems with image content **reproducibility**
   - Problems with “***unknown***” containers
   - Problems with **naming** (app versioning VS release versioning and “**:latest**” confusions)
3. Thinking about solutions to those problems
4. Implementing first batch of solutions *(we are HERE, and not even finished…)*

KEY REQUIREMENTS
for container image creation process

Having global system-wide (technical) process for creating and storing images, satisfying:

- **DEAD SIMPLE to use** for Developers (and docker image creators in general)
- **ACCOUNTABILITY** of actions (knowing who created what)
- **DEPENDABLE** (certainty that what get there wasn’t tampered with)
- **TEAM IS IN CONTROL** (**flexibility** to create, delete update ***any image*** team owns)
- **RBAC SEGREGATION** (only TeamA can CRUD images in RegistryA)
- **IMMUTABLE** (certainty that some image names will stay and keep unchanged forever)
- **VERSIONING UNIFICATION** (**single** versioning and tagging pattern used globally)
- **BASE IMAGE CONTROL** (no proliferation of bazilion changing external base images)
- **IMAGES DOCUMENTED*** (I know from image what it is about and how to use it)
-
- **not-a-requirement:** **JENKINS ONLY** (as Pipeline DSL and Freestyle Jenkins jobs)

Creating Images (commit container)

```
> docker run --name=cont1 -it --entrypoint bash my/favourite-image-name

$$  doing-some-stuff > /inside-container

> docker commit cont1 d.msd.com/my/new-image

> docker push d.msd.com/my/new-image
```

Creating Images (commit container)

Flexibility

Convenience

Speed of “from idea to image”

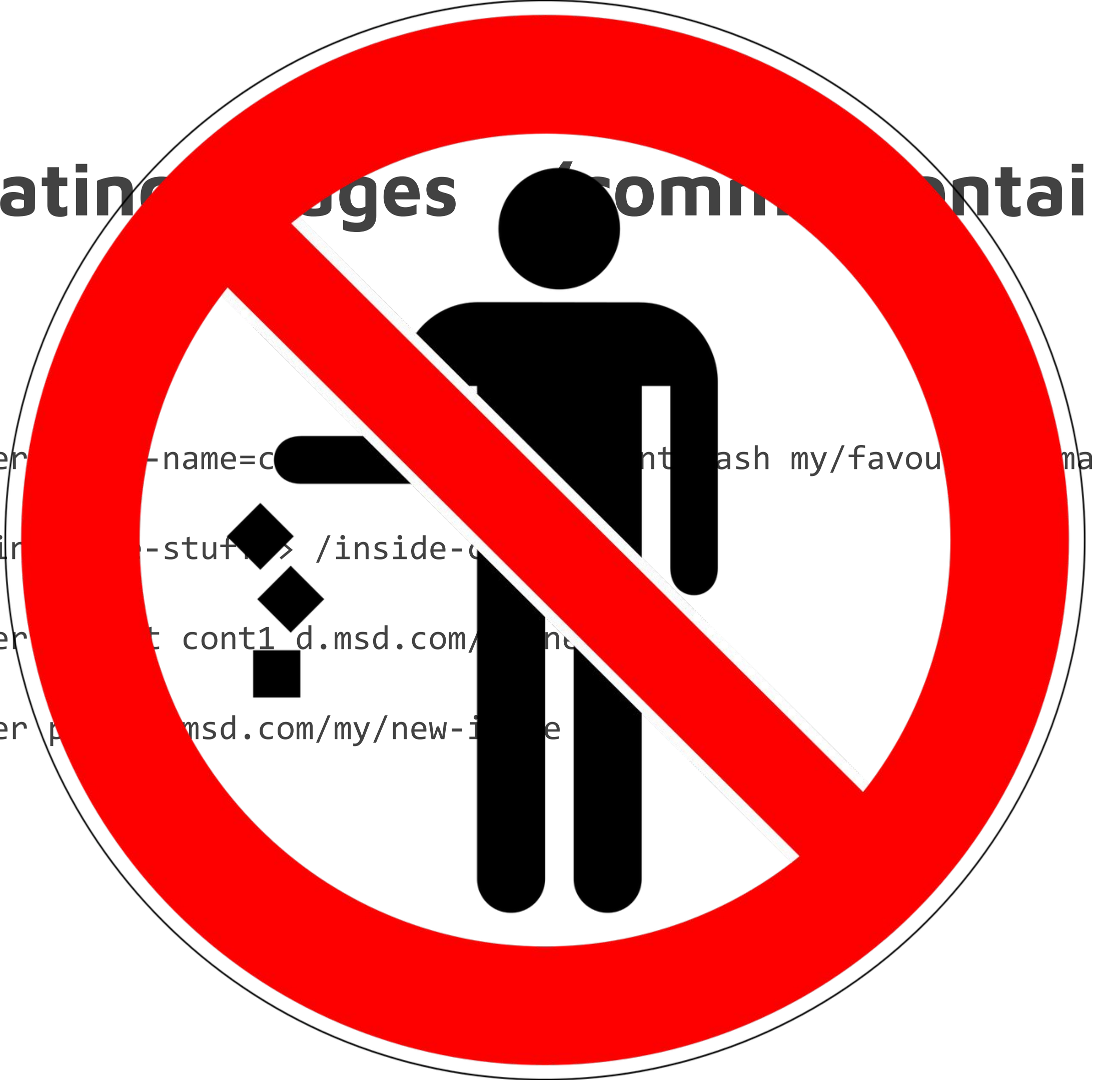
Interactivity

Reproducibility - only 1 person know how image was created (and in two months they don’t remeber as well)

Transparency - same as above, hard to tell what was the purpose of image and near impossible to improve upon it

Automation - interactivity prevents automating solution

Creatin ges omm ntainer)



```
> docker -name=c nt ash my/favou mage-name

$$  doir -stuf > /inside-

> docker t cont1 d.msd.com/ ne

> docker p msd.com/my/new-i e
```

Creating Images (Dockerfile as text)

```
cat << END_OF_DOCKERFILE  >  Dockerfile
  FROM  my/favourite-image-name
  RUN doing-some-stuff > /inside-container
END_OF_DOCKERFILE

> docker build -t d.msd.com/my/new-image .

> docker login -u name -p password d.msd.com

> docker push d.msd.com/my/new-image
```

Creating Images (Dockerfile as text)

Reproducible and **Automated** - *I found this in Jenkins job*

Transparent - everybody can see what’s being done

Change history - only latest version is available

Permanent link - as above (not possible to link to content at some point in the past)

Image name and image content not associated - give content of this Dockerfile to 50 people and tell them to push an image with the content, you will get 50 different image names with the same/similar content

Creatin ges (Docke as text)

```
cat << END KERFILE
  FROM ourite-image-na
  RUN do me-st f > /inside er
END_OF_DO E

> docker -t d. d.com/my/n im

> docker pu sd.com/my/new-i e
```

Creating Images (Dockerfile in GIT 1)

```
FROM  my/favourite-image-name
RUN doing-some-stuff > /inside-container
```

```
> git clone ...

> docker build -t d.msd.com/my/new-image .

> docker push d.msd.com/my/new-image
```

Creating Images (Dockerfile in GIT 1)

Change history

Permanent Link to any version

Image name and image content not associated
- (1 Dockerfile + 50 people = ~50 “same” images in registry under different names)

Creatin ges (Docke in GIT 1)

```
FROM      vourite
RUN d    some-stuff > /i      a er
```

```
> git cl

> docker b    t d.msd.com/my/n im    .

> docker pus    com/my/new-i e
```

Creating Images (Dockerfile in GIT 2)

```
FROM  my/favourite-image-name
RUN doing-some-stuff > /inside-container
LABEL image.name=”d.msd.com/my/image” \
      image.tag=”latest”
```

```
> IMG_ID=$(docker build --pull --quiet .)
> I=$(docker inspect -f '{{ index .Config.Labels "image.name" }}' $IMG_ID)
> T=$(docker inspect -f '{{ index .Config.Labels "image.tag" }}' $IMG_ID)
> docker tag  “$IMG_ID”  “$I:$T”
> docker push “$I:$T”
```

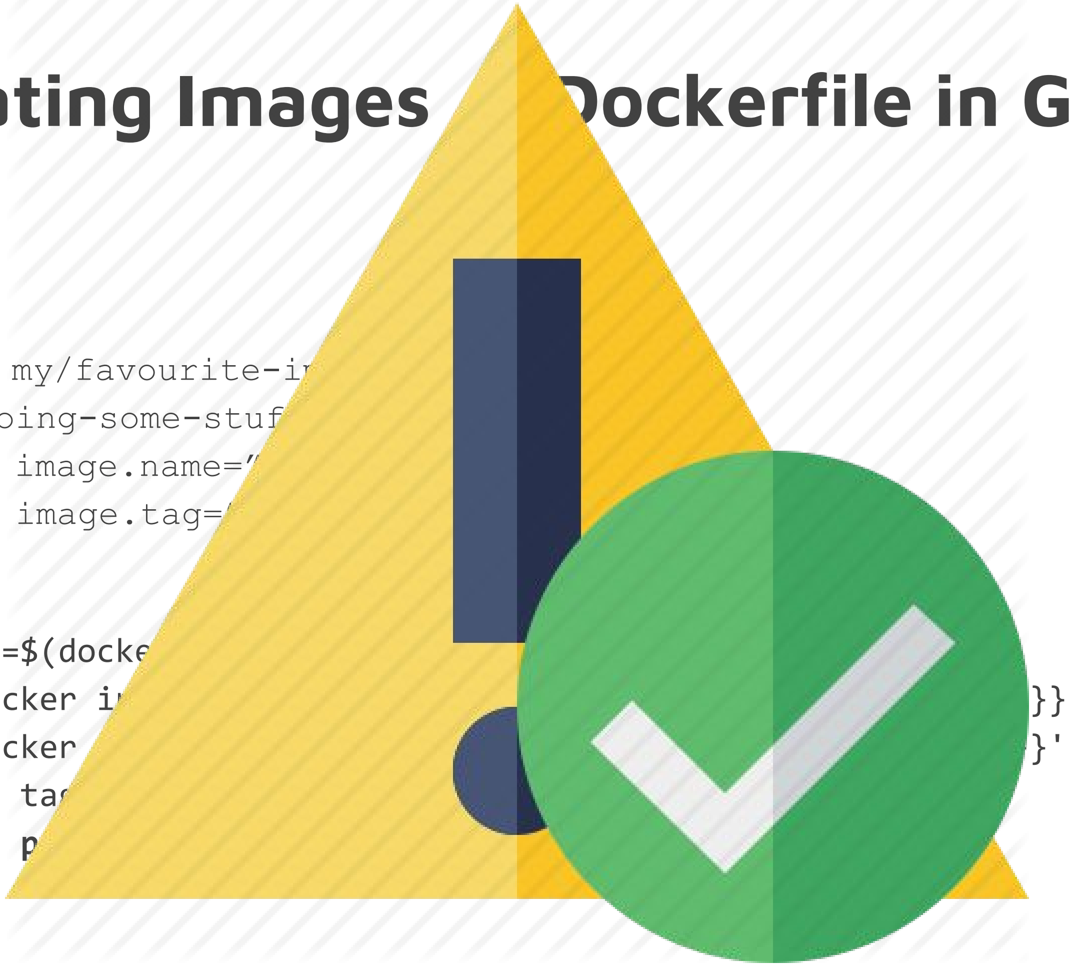
Creating Images   (Dockerfile in GIT 2)

Image name and image content associated

Starting point that can work…

…but still, lot of problems (on different levels)

Creating Images Dockerfile in GIT 2)



```
FROM  my/favourite-i
RUN doing-some-stuf
LABEL image.name="
      image.tag="
```

```
> IMG_ID=$(docke
> I=$(docker i                                            }}' $IMG_ID)
> T=$(docker                                              }' $IMG_ID)
> docker ta
> docker p
```

HARD RULES (...so far)

- Nobody can directly push
- **Dockerfile** from internal GIT repo (Bitbucket Server in our case)
- Name and tag(s) of an image MUST be in Dockerfile, always

→ JENKINS job needs to know this (3 mandatory parameters, this is DEAD SIMPLE)

1. git **repository URL**
2. git **branch** or **pattern**
3. **path** to docker build context
4. filename of dockerfile (OPTIONAL, default: to "**`Dockerfile`**")
5. Credentials under which to push image

1

Dockerfile content

(problems? shout out loud!)

```
FROM   someguy/random-image

MAINTAINER    Rachael Tyrell <rachael.tyrell@msd.com>

RUN    doing-some-stuff > /inside-container

LABEL    image.name=”d.msd.com/myteam/myimage” \
         image.tag=”latest”
```

Dockerfile content

```
FROM    someguy/random-image:v4.3.2      ## still can change any time

MAINTAINER    Rachael Tyrell <rachael.tyrell@msd.com>

RUN    doing-some-stuff > /inside-container

LABEL    image.name=”d.msd.com/myteam/myimage” \
         image.tag=”latest”
```

FROM whatever/image

Convenient and **Fast -** just reuse existing image and boom

Immutability - such base image can be deleted any day and replaced with different image (not under our control)

Dependability - (from above) two consecutive builds of the same Dockerfile might result in 2 different images (content-wise).

Dockerfile content

```
FROM  someguy/random-image@sha256:d0e0a0d0b0e0e0f0ac6cf801996b08abce246a4e0
0f0a0d0e0d0e0c0a0d0e498
                          ## technically correct, but horrible and unusable

MAINTAINER    Rachael Tyrell <rachael.tyrell@msd.com>

RUN    doing-some-stuff > /inside-container

LABEL    image.name=”d.msd.com/myteam/myimage” \
         image.tag=”latest”
```

Dockerfile content

```
FROM    someguy/random-image

MAINTAINER    Rachael Tyrell <rachael.tyrell@msd.com>

RUN    doing-some-stuff > /inside-container

LABEL    image.name=”d.msd.com/myteam/myimage” \
         image.tag=”latest”

## myteam is “DTR concept” of organisation and myimage is image name
## hence: 1 grand registry for everybody with 2 levels (team + image name)
## tag :latest overused (but different understanding for different people)
## how distinguish app version, release version, git version and quality?
```

d.msd.com/myteam/myimage:latest

“**Standard**” naming **convention***

Simple - anybody understands

Flexible - use ***imagename for whatever*** and ***tag also for whatever***

Version omit - psychological issue with naming scheme “organization/**image-name**”

Version mismatch - app version and release version (ex: application Talker v1.0.3, several releases with different image content)

Tag anarchy - using tags for whatever by different teams prevents cross-team understanding of versioning

*) GitHub inspired

Dockerfile content

```
FROM   someguy/random-image

MAINTAINER    Rachael Tyrell <rachael.tyrell@msd.com>

ENV   APP_VERSION=0.66.6

RUN    doing-some-stuff > /inside-container

LABEL    image.name=”myteam.d.msd.com/my/image/name/$APP_VERSION” \
         image.tag=”latest”
```

Dockerfile content

```
FROM   someguy/random-image

MAINTAINER    Rachael Tyrell <rachael.tyrell@msd.com>

ENV   APP_VERSION=0.66.6

RUN    doing-some-stuff > /inside-container

LABEL    image.name=”myteam.d.msd.com/my/image/name/$APP_VERSION” \
         image.tags=”$GIT_SHA1 $GIT_BRANCHES $GIT_TAGS latest”

## still, confusions with meaning of tag “latest”
```

Dockerfile content

```
FROM   someguy/random-image

MAINTAINER    Rachael Tyrell <rachael.tyrell@msd.com>

ENV   APP_VERSION=0.66.6

RUN    doing-some-stuff > /inside-container

LABEL    image.name=”myteam.d.msd.com/my/image/name/$APP_VERSION” \
         image.tags=”$GIT_SHA1 $GIT_BRANCHES $GIT_TAGS current”
```

Dockerfile content

```
FROM   someguy/random-image

MAINTAINER    Rachael Tyrell <rachael.tyrell@msd.com>

ENV   APP_VERSION=0.66.6

RUN    doing-some-stuff > /inside-container

LABEL    image.name=”myteam.d.msd.com/my/image/name/$APP_VERSION” \
         image.tags=”$GIT_SHA1 $GIT_BRANCHES $GIT_TAGS current”
```

Dockerfile content

```
FROM   base-image.d.msd.com/myteam/unibase/rel-2017w43

MAINTAINER    Rachael Tyrell <rachael.tyrell@msd.com>

ENV   APP_VERSION=0.66.6

RUN    doing-some-stuff > /inside-container

LABEL    image.name=”myteam.d.msd.com/my/image/name/$APP_VERSION” \
         image.tags=”$GIT_SHA1 $GIT_BRANCHES $GIT_TAGS current”
```

BASE Dockerfile content

```
FROM   someguy/random-image:v4.3.2

MAINTAINER    Dr. Eldon Tyrell <eldon.tyrell@msd.com>

LABEL    image.name=”base-image.d.msd.com/myteam/unibase/rel-2017w43” \
         image.tags=”$GIT_SHA1           \
                     $GIT_BRANCHES       \
                     $GIT_TAGS           \
                     build-$BUILD_ID     \
                     latest”       ## anyway, what is a version and which?

## approval process for base images, content scan (CVEs, bad practices, …)
```

Dockerfile content

```
FROM   someguy/random-image:v4.3.2

MAINTAINER    Rachael Tyrell <rachael.tyrell@msd.com>

ENV   APP_VERSION=0.66.6

RUN    doing-some-stuff > /inside-container

LABEL    image.name=”myteam.d.msd.com/my/image/name/$APP_VERSION” \
         image.tags=”$GIT_SHA1 $GIT_BRANCHES $GIT_TAGS current”
```

Dockerfile content

```
FROM   base-image.d.msd.com/myteam/unibase/rel-2017w43

MAINTAINER    Rachael Tyrell <rachael.tyrell@msd.com>

ENV   APP_VERSION=0.66.6

RUN    doing-some-stuff > /inside-container

LABEL    image.name=”myteam.d.msd.com/my/image/name/$APP_VERSION” \
         image.tags=”$GIT_SHA1 $GIT_BRANCHES $GIT_TAGS current”
```

2

HARD RULES (...so far)

- Nobody can directly push
- Jenkins reads **Dockerfile** from internal GIT repo (Bitbucket Server in our case)
- Name of an image MUST be in Dockerfile, always
- Multiple labels, used for tracking git or jenkins build (image *development* focus)
- Special registry for “base-images” with approval gates and naming scheme
- Images can be based ***only*** from internal images (no external bases)
- Abandon “one global registry” - every team gets own docker registry

REQUIREMENTS for container Images (recapitulation)

Having global system-wide (technical) process for creating and storing images, satisfying:

- **ACCOUNTABLE** actions (knowing who created what)
- **DEPENDABLE** (certainty that what get there wasn't tampered with)
- **BASE IMAGE CONTROL** (no proliferation of bazilion changing external base images)
- **VERSIONING UNIFICATION** (**single** versioning and tagging pattern used globally)
- **DEAD SIMPLE to use** for Developers (and docker image creators in general)
- **RBAC SEGREGATION** (only TeamA can CRUD images in RegistryA)
- **IMAGES DOCUMENTED** (I know from image what it is about and how to use it)
- **TEAM IS IN CONTROL** (**flexibility** to create, delete update **any image** team owns)
- **IMMUTABLE** (certainty that some image names will stay and keep unchanged forever)

REQUIREMENTS for container Images (recapitulation)

Having global system-wide (technical) process for creating and storing images, satisfying:

- **TEAM IS IN CONTROL** (**flexibility** to create, delete update **any image** team owns)
- **IMMUTABLE** (certainty that some image names will stay and keep unchanged forever)

How to reconcile those two requirements?

SOLUTION: Split 1 docker registry in 2 (or multiple ones)

Every team will get **two** (or more) registries instead of one

```
myteam-dev.d.msd.com     (dev registry)
myteam.d.msd.com         (prod registry)
```

REQUIREMENTS for container Images (recapitulation)

Having global system-wide (technical) process for creating and storing images, satisfying:

- **TEAM IS IN CONTROL** (**flexibility** to CRUD **any image** in DEV registry)
- **IMMUTABLE** (**certainty** that images in PROD registry will stay forever same)

Team has total control over **DEV repo** and any team member can push images (via Jenkins) or delete them (via API or UI to Artifactory).

Nobody can push directly to **PROD repo**.
Only “good” images from DEV repo can be ***promoted*** to production registry (**@sha256** kept).
Then it will stay in that PROD registry under that given name FOREVER.
The name is taken since then and can’t be **ever** used by any different image
(meaning: **update / overwrite are NOT possible**).

REQUIREMENTS for container Images (recapitulation)

Having global system-wide (technical) process for creating and storing images, satisfying:

- **TEAM IS IN CONTROL** (**flexibility** to CRUD **any image** in DEV registry)
- **IMMUTABLE** (**certainty** that images in PROD registry will stay forever same)
- **VERSIONING UNIFICATION** (**single** versioning and tagging pattern used globally)

Can we unify naming scheme across all registries?

REQUIREMENTS for container Images (recapitulation)

Having global system-wide (technical) process for creating and storing images, satisfying:

- **TEAM IS IN CONTROL** (**flexibility** to CRUD **any image** in DEV registry)
- **IMMUTABLE** (**certainty** that images in PROD registry will stay forever same)
- **VERSIONING UNIFICATION** (**single** versioning and tagging pattern used globally)

Can’t be done *easily*…

…unless some expectation will be broken…

(*opinionated* new standard)

SOLUTION: Redefine tag **latest**

- For **DEV** registry: any tag **except** “**`latest`**” is allowed
  no permission to push an image with tag “latest” - solved by Artifactory

- For **PROD** registry: **only** tag “**`latest`**” is allowed
  permission to promote images that can be only tagged “latest” - solved by Artifactory

SOLUTION: Redefine tag **latest**

- For **DEV** registry: any tag **except** “**`latest`**” is allowed
  no permission to push an image with tag “latest” - solved by Artifactory

- For **PROD** registry: **only** tag “**`latest`**” is allowed
  permission to promote images that can be only tagged “latest” - solved by Artifactory

New shared semantics (common company-wide shared understanding of “**:latest**” tag):

1. Whenever I see **image name without a tag, it is always production image**.
   example: ***rainbow.d.msd.com/progeny/rel-2017w23***
2. **Development image must always have a tag** in a name.
   example: rainbow**-dev**.d.msd.com/progeny:***feature_sync-css-styles-to-company-teal***
3. **Suffix “-dev” → always tag. Without “-dev” suffix → never tag.**
   These can’t mismatch.
   wrong examples: ~~**rainbow**.d.msd.com/progeny**:something**~~
   ~~**rainbow-dev**.d.msd.com/progeny~~

SOLUTION: Redefine tag **latest**

```
FROM   someguy/random-image:v4.3.2

MAINTAINER    Dr. Eldon Tyrell <eldon.tyrell@msd.com>

LABEL    image.name=”base-image-dev.d.msd.com/myteam/unibase” \
         image.tags=”$GIT_SHA1           \
                     $GIT_BRANCHES       \
                     $GIT_TAGS           \
                     build-$BUILD_ID     \
                     4.3.2               \
                     latest”       ## anyway, what is a version and which?
```

SOLUTION: Redefine tag **latest**

```
FROM   someguy/random-image:v4.3.2

MAINTAINER    Dr. Eldon Tyrell <eldon.tyrell@msd.com>

LABEL    image.name=”base-image-dev.d.msd.com/myteam/unibase-4.3.2” \
         image.tags=”$GIT_SHA1          \
                     $GIT_BRANCHES      \
                     $GIT_TAGS          \
                     build-$BUILD_ID    \
                     current”
```

SOLUTION: Redefine tag **latest**

```
FROM   someguy/random-image:v4.3.2

MAINTAINER    Dr. Eldon Tyrell <eldon.tyrell@msd.com>

LABEL    image.name=”base-image-dev.d.msd.com/myteam/unibase-4.3.2” \
         image.tags=”$GIT_SHA1          \
                    $GIT_BRANCHES      \
                    $GIT_TAGS          \
                    build-$BUILD_ID    \
                    current”

--------------------- IMAGE NAMES THAT WILL BE PUSHED ---------------------
base-image-dev.d.msd.com/myteam/unibase-4.3.2:28c6839911fc0df72ec6bd62fa91b5c3703f4f43
base-image-dev.d.msd.com/myteam/unibase-4.3.2:master
base-image-dev.d.msd.com/myteam/unibase-4.3.2:build-MY_TEAM_BASE-21
base-image-dev.d.msd.com/myteam/unibase-4.3.2:current
```

SOLUTION: Redefine tag **latest**

```
--------------------- IMAGE NAMES THAT WERE PUSHED -----------------------
base-image-dev.d.msd.com/myteam/unibase-4.3.2:28c6839911fc0df72ec6bd62fa91b5c3703f4f43
base-image-dev.d.msd.com/myteam/unibase-4.3.2:master
base-image-dev.d.msd.com/myteam/unibase-4.3.2:build-MY_TEAM_BASE-21
base-image-dev.d.msd.com/myteam/unibase-4.3.2:current
```

SOLUTION: Redefine tag **latest**

```
--------------------- IMAGE NAMES THAT WERE PUSHED -----------------------
base-image-dev.d.msd.com/myteam/unibase-4.3.2:28c6839911fc0df72ec6bd62fa91b5c3703f4f43
base-image-dev.d.msd.com/myteam/unibase-4.3.2:master
base-image-dev.d.msd.com/myteam/unibase-4.3.2:build-MY_TEAM_BASE-21
base-image-dev.d.msd.com/myteam/unibase-4.3.2:current
```

Person with proper access rights can now promote DEV image to be a PROD image:

```
base-image-dev.d.msd.com/myteam/unibase-4.3.2:28c6839911fc0df72ec6bd62fa91b5c3703f4f43
                                     ↓
                                     ?
```

SOLUTION: Redefine tag **latest**

```
---------------------- IMAGE NAMES THAT WERE PUSHED ------------------------
base-image-dev.d.msd.com/myteam/unibase-4.3.2:28c6839911fc0df72ec6bd62fa91b5c3703f4f43
base-image-dev.d.msd.com/myteam/unibase-4.3.2:master
base-image-dev.d.msd.com/myteam/unibase-4.3.2:build-MY_TEAM_BASE-21
base-image-dev.d.msd.com/myteam/unibase-4.3.2:current
```

Person with proper access rights can now promote DEV image to be a PROD image:

```
base-image-dev.d.msd.com/myteam/unibase-4.3.2:28c6839911fc0df72ec6bd62fa91b5c3703f4f43
                                         ↓
             base-image.d.msd.com/myteam/unibase-4.3.2/rel-2017w43:latest
```

SOLUTION: Redefine tag **latest**

```
base-image-dev.d.msd.com/myteam/unibase-4.3.2:28c6839911fc0df72ec6bd62fa91b5c3703f4f43
                                       ↓
         base-image.d.msd.com/myteam/unibase-4.3.2/rel-2017w43:latest
```

SOLUTION: Redefine tag **latest**

```
base-image-dev.d.msd.com/myteam/unibase-4.3.2:28c6839911fc0df72ec6bd62fa91b5c3703f4f43
                                  ↓
base-image.d.msd.com/myteam/unibase-4.3.2/rel-2017w43:latest


base-image-dev.d.msd.com/myteam/unibase-4.3.2:34468d9ce11743878893acc9b566c8873c0bb04c
                                  ↓
base-image.d.msd.com/myteam/unibase-4.3.2/rel-2017w43.1:latest
```

SOLUTION: Redefine tag **latest**

base-image-dev.d.msd.com/myteam/unibase-4.3.2**:28c6839911fc0df72ec6bd62fa91b5c3703f4f43**

↓

base-image.d.msd.com/myteam/unibase-4.3.2**/rel-2017w43**:latest

base-image-dev.d.msd.com/myteam/unibase-4.3.2**:34468d9ce11743878893acc9b566c8873c0bb04c**

↓

base-image.d.msd.com/myteam/unibase-4.3.2**/rel-2017w43.1**:latest

base-image-dev.d.msd.com/myteam/unibase-4.3.2**:baed8b86aff3408af903d303c25ff6290af088c3**

↓

base-image.d.msd.com/myteam/unibase-4.3.2**/rel-2017w44**:latest

SOLUTION: Redefine tag **latest**

Immutable name forever:

```
base-image.d.msd.com/myteam/unibase-4.3.2/rel-2017w43:latest
base-image.d.msd.com/myteam/unibase-4.3.2/rel-2017w43.1:latest
base-image.d.msd.com/myteam/unibase-4.3.2/rel-2017w44:latest
```

SOLUTION: Redefine tag **latest**

Immutable name forever:

```
base-image.d.msd.com/myteam/unibase-4.3.2/rel-2017w43:latest
base-image.d.msd.com/myteam/unibase-4.3.2/rel-2017w43.1:latest
base-image.d.msd.com/myteam/unibase-4.3.2/rel-2017w44:latest
```

But overwrite is sometimes still convenient:

```
base-image.d.msd.com/myteam/unibase-4.3.2/latest:latest
```

3

REQUIREMENTS for container Images (recapitulation)

Having global system-wide (technical) process for creating and storing images, satisfying:

- **ACCOUNTABLE** actions (knowing who created what)
- **DEPENDABLE** (certainty that what get there wasn’t tampered with)
- **BASE IMAGE CONTROL** (no proliferation of bazilion changing external base images)
- **VERSIONING UNIFICATION** (**single** versioning and tagging pattern used globally)
- **TEAM IS IN CONTROL** (**flexibility** to create, delete update ***any image*** team owns)
- **IMMUTABLE** (certainty that some image names will stay and keep unchanged forever)
- **DEAD SIMPLE to use** for Developers (and docker image creators in general)
- **RBAC SEGREGATION** (only TeamA can CRUD images in RegistryA)
- **IMAGES DOCUMENTED** (I know from image what it is about and how to use it)

Docker login & push

```
> docker login -u NAME -p PASSWORD      myteam-dev.d.msd.com

> docker push                           myteam-dev.d.msd.com/stuff


> jq -r '.auths["myteam-dev.d.msd.com"].auth' ~/.docker/config.json | base64 -d
NAME:PASSWORD
```

Docker login & push

Convenient for single user - login once, don’t bother later

Insecure as hell

Can’t be used in shared setup (JENKINS)

On JENKINS: Once somebody is logged, then everybody is logged

SOLUTION: not using docker client for pushing

Every team have its own service user with privileges to push to their registry.
Jenkins Folders + Jenkins Credentials → Jenkins job expects team to create their own Credentials with name “**docker-push-credential**”.

Single python executable **`docker-push`** that utilizes **docker-py.**
Program expects 2 ENV variables for NAME and PASSWORD, then AuthN + push.
No “login step” needed.

Jenkins job fetch credential “**docker-push-credential**” and pre-fills values to two environment variables and then calls “`docker-push <image-name>`” and it does the job in secure way.

REQUIREMENTS for container Images
(recapitulation)

Having global system-wide (technical) process for creating and storing images, satisfying:

- **ACCOUNTABLE** actions (knowing who created what)
- **DEPENDABLE** (certainty that what get there wasn't tampered with)
- **BASE IMAGE CONTROL** (no proliferation of bazilion changing external base images)
- **VERSIONING UNIFICATION** (**single** versioning and tagging pattern used globally)
- **TEAM IS IN CONTROL** (**flexibility** to create, delete update ***any image*** team owns)
- **IMMUTABLE** (certainty that some image names will stay and keep unchanged forever)
- **RBAC SEGREGATION** (only TeamA can CRUD images in RegistryA)
- **DEAD SIMPLE to use** for Developers (and docker image creators in general)
- **IMAGES DOCUMENTED** (I know from image what it is about and how to use it)

Jenkins pipeline example

(custom shared pipeline library)

```
pipeline {
 agent ‘docker-builder’
 stages {
  stage(“git”) {
   git “https://github.com/our/dockerfiles”, branch: “feature/example”
  }
  stage(“push two docker images”) {

   pushDockerfile “docker/jira”      ## ← expected usage like that

   pushDockerfile(path: “docker/confluence”,
                  file: “confluence-6.4.1.Dockerfile”,
                  cred: “my-custom-credential”)
}}}
```

4

Image documentation

Inspired by http://label-schema.org/rc1/

Simply **require** some set of labels to be specified.
If **labels not present**, **fail** the build.

Image documentation

Inspired by http://label-schema.org/rc1/

Simply **require** some set of labels to be specified.
If **labels not present**, **fail** the build.

- **Mandatory labels** (fail if any is missing)
- **Recommended labels** (don’t fail if this is missing, but complain to user)
- **Labels** added automatically **during the build**
- **ARGS** added automatically **during the build**

```
FROM  base-image.d.msd.com/ubuntu/16.04/rel-2017w39

ENV  CONFLUENCE_VERSION=6.4.2


RUN  what ...
RUN  ever ...


LABEL   PREFIX.image.name=stack-dev.d.msd.com/confluence/${CONFLUENCE_VERSION}  \
        PREFIX.image.tags="current  $GIT_SHA1  $BRANCHES_AND_TAGS  $GIT_TAG_ANNOTATED"  \
        PREFIX.description="Customized Confluence v${CONFLUENCE_VERSION}  (DevOps Stack)"
        ...
```

```
LABEL   PREFIX.image.name=stack-dev.d.msd.com/confluence/${CONFLUENCE_VERSION}  \
        PREFIX.image.tags="current $GIT_SHA1 $BRANCHES_AND_TAGS $GIT_TAG_ANNOTATED"  \
        PREFIX.description="Customized Confluence v${CONFLUENCE_VERSION}  (DevOps Stack)"
        ...
```

```
LABEL   PREFIX.image.name=stack-dev.d.msd.com/confluence/${CONFLUENCE_VERSION}  \
        PREFIX.image.tags="current $GIT_SHA1 $BRANCHES_AND_TAGS $GIT_TAG_ANNOTATED"  \
        PREFIX.description="Customized Confluence v${CONFLUENCE_VERSION}  (DevOps Stack)"
        PREFIX.maintainer.isid="hedl" \
        PREFIX.maintainer.name="Mira Hedl" \
        PREFIX.maintainer.email="gic-devops-stack-admins@msd.com" \
        PREFIX.environment=Production  \
        PREFIX.org.division="Global Software Engineering Competency Center"  \
        PREFIX.org.team="SW Engineering Foundations"  \
        PREFIX.git.dockerfile="https://$GIT_URL/confluence-${CONFLUENCE_VERSION}.Dockerfile?at=$GIT_SHA1"  \
        PREFIX.git.commit=$GIT_SHA1 \
        ...
```

```
LABEL   PREFIX.image.name=stack-dev.d.msd.com/confluence/${CONFLUENCE_VERSION}  \
        PREFIX.image.tags="current $GIT_SHA1 $BRANCHES_AND_TAGS $GIT_TAG_ANNOTATED"  \
        PREFIX.description="Customized Confluence v${CONFLUENCE_VERSION}  (DevOps Stack)"
        PREFIX.maintainer.isid="hedl" \
        PREFIX.maintainer.name="Mira Hedl" \
        PREFIX.maintainer.email="gic-devops-stack-admins@msd.com" \
        PREFIX.environment=Production  \
        PREFIX.org.division="Global Software Engineering Competency Center"  \
        PREFIX.org.team="SW Engineering Foundations"  \
        PREFIX.git.dockerfile="https://$GIT_URL/confluence-${CONFLUENCE_VERSION}.Dockerfile?at=$GIT_SHA1"  \
        PREFIX.git.commit=$GIT_SHA1 \
        \
        PREFIX.params.APR_DISABLED="Disables APR Native library in Apache Tomcat"  \
        PREFIX.params.CROWD_ENDPOINT="URL to crowd endpoint."  \
        PREFIX.params.CROWD_ENDPOINT_application_name="Crowd Application - username"  \
        PREFIX.params.CROWD_ENDPOINT_application_password="Crowd Application - password"  \
        PREFIX.params.SERVER_XML_PROXY="URL to reverse-proxy server if Confluence runs behind proxy"  \
        PREFIX.params.SERVER_XML_maxThreads="Set max number of JVM threads [default: 48]"  \
        PREFIX.params.CATALINA_EXTRA_OPTS="Extra options to Catalina"  \
        PREFIX.params.JAVA_MEM_MS="Minimum JVM heap memory [default: $JAVA_MEM_MS]"  \
        PREFIX.params.JAVA_MEM_MX="Maximum JVM heap memory [default: $JAVA_MEM_MX]"  \
        PREFIX.params.JMX_REMOTE_PORT="Enable JMX on given port (number between 1025 and 65535)"  \
        ...
```

Build-time ARGS and labels

```
> docker build --pull --silent --squash                                             \
               -f “$file_name”                                                      \
               \
               --build-arg GIT_SHA1=$(git rev-parse HEAD)                           \
               --build-arg BRANCHES_AND_TAGS="$(echo ${BRANCHES_AND_TAGS[@]})"      \
               --label PREFIX.build.jenkins_url=${BUILD_URL}                        \
               --label PREFIX.build.date=$(date -R)                                 \
         . . .
```

Meta-data using labels

Labels are **intended for this**

You can append label during build

Always present on Image and Containers - docker inspect

Other services can generate “**doc stickers**” with just image name given (fixed schema)

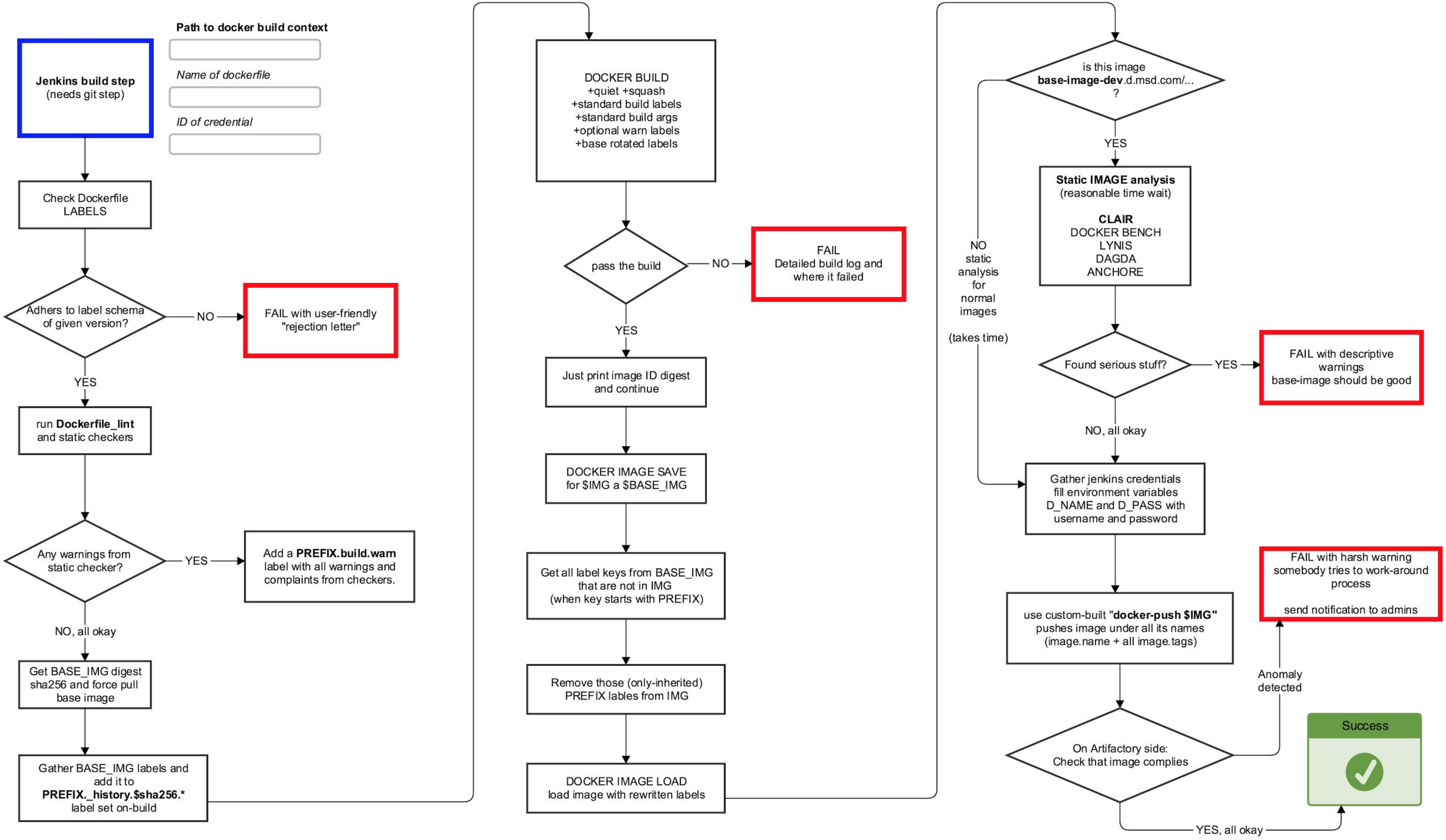
Label inheritance gets in a way - everything is inherited from BASE image, which is good for most cases, but in this special case it is NOT wanted

Overwrites labels from BASE - It would be nice to have labels history (see image base-lineage with all

SOLUTION: label rewrite

```
> BASE_IMG=$(get-docker-base Dockerfile)
> docker pull $BASE_IMG

> IMG=$(docker build --quiet … -f Dockerfile)

## unpack image and base-image into separate directories
> docker image save $BASE_IMG | tar -x -f - -C  my_base
> docker image save $IMG | tar -x -f - -C  my_image

> rewrite-tags --dir=my_image  --base=my_base --inplace
> pushd my_image; tar cf - . | docker image load

## push image under all names/tags
> for tag in $(get_tags $IMG); do
    D_USER=user D_PASS=pass  docker-push “$(get_img $IMG):$tag”
  done
```

5



Jenkins build step
(needs git step)

Path to docker build context

Name of dockerfile

ID of credential

Check Dockerfile LABELS

Adhers to label schema of given version?

NO

FAIL with user-friendly "rejection letter"

YES

run **Dockerfile_lint** and static checkers

Any warnings from static checker?

YES

Add a **PREFIX.build.warn** label with all warnings and complaints from checkers.

NO, all okay

Get BASE_IMG digest sha256 and force pull base image

Gather BASE_IMG labels and add it to **PREFIX._history.$sha256.*** label set on-build

DOCKER BUILD
+quiet +squash
+standard build labels
+standard build args
+optional warn labels
+base rotated labels

pass the build

NO

FAIL
Detailed build log and where it failed

YES

Just print image ID digest and continue

DOCKER IMAGE SAVE for $IMG a $BASE_IMG

Get all label keys from BASE_IMG that are not in IMG (when key starts with PREFIX)

Remove those (only-inherited) PREFIX lables from IMG

DOCKER IMAGE LOAD load image with rewritten labels

is this image **base-image-dev**.d.msd.com/... ?

NO static analysis for normal images

(takes time)

YES

Static IMAGE analysis
(reasonable time wait)

CLAIR
DOCKER BENCH
LYNIS
DAGDA
ANCHORE

Found serious stuff?

YES

FAIL with descriptive warnings base-image should be good

NO, all okay

Gather jenkins credentials fill environment variables D_NAME and D_PASS with username and password

use custom-built "**docker-push $IMG"** pushes image under all its names (image.name + all image.tags)

On Artifactory side: Check that image complies

Anomaly detected

FAIL with harsh warning somebody tries to work-around process

send notification to admins

YES, all okay

Success

✓

That's All Folks!

Questions?

Touch points…

(not planned yet)

- **Static analysis - many tools here:** https://sysdig.com/blog/20-docker-security-tools/

  Clair scanner for static analysis BEFORE? Pushing

  OpenSCAP (atomic scan)

  Banyan Collector (https://github.com/banyanops/collector)

  Docker Bench for Security, whitelist what you don’t need

  Lynis (https://github.com/CISOfy/lynis) - lynis audit dockerfile FileName,

  Dagda (https://github.com/eliasgranderubio/dagda)

  Anchore (https://github.com/anchore/anchore)

- **jFrog XRay + BlackDuck / Aqua / Snyk**

- **Notary & Docker Content Trust**
- **dockerfile_lint**