Docker Implementation
Version 1

TRIUMF ATLAS Tier-1

by

Denice Deatrich

Last update:

2016-08-12

Index

1 Introduction

The time has come to investigate using Docker, a Containers-based technology, for worker node deployment. Worker nodes, because of their uniformity in configuration and requirements, are good candidates for Containers. With KVM virtualization for example, there is much more overhead in process, memory and disk space running KVM guests for each worker. With Containers a much more light-weight environment is needed to sustain a worker implementation – system-wide changes are shared – only the application is sand-boxed.

This document summarizes the initial setup and experience using Docker for a worker node infrastructure. As always at the Tier-1 we like to initially investigate new technologies assuming a 'scratch' implementation, where we start from the basics to better understand it. Thus, though I show how to access external Docker images, I focus instead on building and deploying our own.

Because we now use Ansible to manage server configurations this document will explain deployment by showing both the commands and configurations needed, as well as some associated Ansible role-based task snippets.

The Docker infrastructure documented here includes installation, configuration and run-time examples for:

- Docker Engine – the software running on the host that sponsors containers
- Docker Registry/Distribution – software that provides storage, searching and retrieval of docker images
- Image building – how to build you own container images
- Docker Containers – run-time considerations

2 Getting Docker

The Red Hat implementation of Docker is not deployed in the mainstream installation base. Instead it is made available by Red Hat in their 'extras' channel – this description is on their web site[1]:

> *Red Hat is introducing a new channel in the Red Hat Customer Portal for Red Hat Enterprise Linux called "Extras." The Extras channel is intended to give customers access to select, rapidly evolving technologies. These technologies may be updated more frequently than they would otherwise be in a Red Hat Enterprise Linux minor release. The technologies delivered in the Extras channel are fully supported.*
>
> *Over time, these technologies will continue to mature and stabilize and may eventually be added to the Red Hat Enterprise Linux channel to which the Red Hat Enterprise Linux life-cycle policies apply.*

1 https://access.redhat.com/support/policy/updates/extras

Note that this does *not* guarantee that the RHEL implementation of Docker will always be there. At this point in time, Docker for RHEL **7** is re-built and published by both CentOS and Scientific Linux (SL). As the CentOS versions are more recent they have been mirrored by us, and the software can be installed at the Tier-1 using our ***t1-release-rh-extras*** RPM.

This document is based on our experience running Docker on SL; however it applies equally to a CentOS experience.

Docker Engine RPMs for SL and CentOS **6** can be found in EPEL, however they are much older versions, and have no supported path forward[1]. However Docker on SL7 systems can run both SL6 and SL7 application containers. You give up some disk space in running SL6 containers on an SL7 engine because you necessarily need to install SL6-based supporting libraries for any SL6 container.

3 Pre-install Checklist

3.1 Storage preparation

The default behaviour of the Red Hat RPMs is to use loopback devices if you have made no provision for storage for your containers. Loopback devices should not be used in production; you will get a warning when you use them:

> *Usage of loopback devices is strongly discouraged for production use. Either use `--storage-opt dm.thinpooldev` or use `--storage-opt dm.no_warn_on_loop_devices=true` to suppress this warning.*

Moreover, the server should be configured with the necessary *spare* disk space, either as complete LUNs or devices, or as complete volume groups. In this document I use a spare volume group named ***vg1*** for our thin-provisioned storage.

Another issue is local docker container image storage. Each *worker* docker engine may have a number of pulled local images for testing and validation. Rather than storing these in the default location, /var/lib/docker, I decide to locate them in a specific directory not used by the OS, with adequate space. It is mounted as ***/docker***

3.2 Network setup

I did not want to use the default bridging setup on **172.17.0.0/16** provided by the docker RPMs – this is an issue to revisit later. Instead I decided to bridge to our existing private network on **10.0.0.0/16**. To that end I configure the bridge on the Docker engine node ahead of time. In this document the name of the bridge is ***br0***. At this early phase of testing I limit the range of IP ad-

1 https://access.redhat.com/solutions/1378023

dresses for each blade chassis to the local /24 subnet range – e.g. 10.0.3.0/24; and then further restrict using the fixed CIDR-range option to a unique /28 group in that /24 group. We need to avoid address clashes with addresses used in OpenStack testing on the same test blade chassis.

4 Installation

4.1 Docker Engine installation and configuration

For the Tier-1 I install the local release rpm on the Docker Engine worker node host for the Red Hat Extras repository, and then install the docker packages:

```
# yum install t1-release-rh-extras
# yum –enablerepo=rh-extras install docker atomic
```

A few configuration files need to be modified before you enable and start docker in daemon mode.

These files need to be modified to address issues specific to our data centre:

4.1.1 /etc/sysconfig/docker-network

```
## this is just an example for this test blade
# diff docker-network docker-network.original
2,3c2
< DOCKER_NETWORK_OPTIONS="--ipv6=false --mtu=9000 –bridge=br0 --fixed-
cidr=10.0.3.144/28"
<
---
> DOCKER_NETWORK_OPTIONS=
```

- I want a customized network bridge setup
- IPv6 is disabled at this time
- You need to explicitly give the MTU if it is not the default 1500

4.1.2 /etc/sysconfig/docker-storage-setup

```
# diff docker-storage-setup docker-storage-setup.original
5,7d4
<
< VG='vg1'
<
```

- I use the free volume group here

4.1.3 /etc/sysconfig/docker

```
# diff docker docker.original
4c4
< OPTIONS='--selinux-enabled=false -g /docker'
---
> OPTIONS='--selinux-enabled'
13d12
< ADD_REGISTRY='--add-registry pps04.lcg.triumf.ca:5000'
19c18
< BLOCK_REGISTRY='--block-registry docker.io'
---
> # BLOCK_REGISTRY='--block-registry'
25d23
< INSECURE_REGISTRY='--insecure-registry pps04.lcg.triumf.ca:5000'
34c32
< DOCKER_TMPDIR=/var/tmp
---
> # DOCKER_TMPDIR=/var/tmp
44,47d41
```

- during initial testing I turn SELinux off
- I move the root of the docker daemon runtime to */docker* ; as already mentioned the default otherwise is */var/lib/docker*. We need some space to grow while we understand how many images we need to keep locally on the node.
- I block access to images at docker.io[1]. Instead we will use a local, private registry running on host pps04.lcg.triumf.ca which is firewalled off from the world. By default Docker registries run on port 5000, but can obviously run on any unprivileged port. Initially the registry host will not be using a secure protocol; thus it must be explicitly labelled *insecure* in the configuration before a Docker engine can successfully interact with it.
- I switch to using /var/tmp for temporary files; the default temporary file location would otherwise be /var/lib/docker/tmp/

Once these files are changed I can enable and start docker:

```
# systemctl enable docker
# systemctl start docker
```

4.1.3.1 External Images

For the Tier-1 data centre, our light-path nodes cannot get to the commerical network – docker.io or redhat.com - without a proxy. This is possible in the /etc/sysconfig/docker with a setting like:

https_proxy=somehost.triumf.ca:3130

However, this also confuses your local pulls, which I believe try to use the proxy to get all images.

1 https://docs.docker.com/docker-hub/repos/

5 Docker Registry installation and configuration

There are two branches of software that provide a Docker Registry at this time:

1. The legacy registry which you get when you install the *docker-registry* RPM
2. The next generation registry provided by the *docker-distribution* RPM

Since the Docker 'distribution' version currently provided by Red Hat in the extras repository seems to be missing some important functionality, like a working search interface in simple setups, I opted to set up a legacy registry in order to forge ahead. However, I created an Ansible role for Docker Distribution during testing; see:

https://git.lcg.triumf.ca/?p=ansible;a=tree;f=roles/docker-distribution

We will try it again in the near future.

5.0.1 Docker Registry

I use host pps04.lcg.triumf.ca for an initial local test registry. The registry daemon is configured to run on the default port, 5000. Only data centre nodes are allowed to access this port. Before installing the software a file system is created with enough growable space for the test environment. The mount point is named */data/docker/* and it will house the container images.

Install the RPM:

```
# yum –enablerepo=rh-extras install docker-registry
```

Before enabling and starting the daemon I modify the its configuration files to suit our environment:

5.0.2 /etc/sysconfig/docker-registry

```
# diff docker-registry docker-registry.original
< REGISTRY_ADDRESS=206.12.1.140
---
> REGISTRY_ADDRESS=0.0.0.0
```

- I specifically list the IP address of this host, since I had another network attached to this host in the early testing. For a single-homed host this would not be needed.

5.0.3 /etc/docker-registry.yml

There are many possible 'flavours' of Docker registries, and its default YAML configuration list example configurations for all types. To simply our presentation I removed them all except the

default local, unsecured flavour. The complete file is available in the appendix of this document. Noteworthy settings are:

Search index area – the index for the search function is moved from /tmp into the data area – specifically /data/docker/index/

```
<     sqlalchemy_index_database: _env:SQLALCHEMY_INDEX_DATABASE:sqlite:////data/docker/index/docker-registry.db
---
>     sqlalchemy_index_database: _env:SQLALCHEMY_INDEX_DATABASE:sqlite:////tmp/docker-registry.db
```

Storage path for images – the default is /var/lib/registry; I use instead the data area, specifically /data/docker/registry/

```
<     storage_path: _env:STORAGE_PATH:/data/docker/registry
---
>     storage_path: _env:STORAGE_PATH:/var/lib/docker-registry
```

Once these files are changed I can enable and start docker-registry:

```
# systemctl enable docker-registry
# systemctl start docker-registry
```

6 Image building

I used another test node already installed with SL6 to build some SL images for containers. A test build system should be considered expendable, since mistakes in building test images might overwrite files or render it inoperable – do not use a production system for this purpose.

The Docker 'contrib' area on github has a number of example scripts that can be used to build images, depending on your underlying build tools. For our needs I modified the 'mkimage-yum.sh[1]' script. Our modified script is available at the URL[2] in the footnote. It essentially creates an image in a few yum stanzas into a target directory using yum. A few of the options for yum are important, as they allow us to install into an alternate root, selecting only mandatory packages without accompanying documentation; they are:

- —releasever
- —installroot
- --setopt=tsflags=nodocs
- --setopt=group_package_types=mandatory

The script writes into a target directory, which is tarred and gzipped at the end. It is this tarball which is copied to a working directory on the registry node, where it is later imported into the registry.

1 https://github.com/docker/docker/blob/master/contrib/mkimage-yum.sh
2 http://grid.triumf.ca/share/

The script first sets up the bare minimum number of device files in the target area. Note that the Tier-1 release RPMs use a yum variable that determines if the mirror is accessed by NFS or by HTTP. For a docker image we want to avoid NFS since it introduced complexity in the container setup, so the variable gets set to an HTTP URL at the end of the script.

6.1 Yum stanzas

These are descriptions of the yum stanzas invoked in the script. By breaking it down we can inspect disk space used so far at each step.

1. Because we wish to use the local mirror for our installation I first install the needed Tier-1 release RPMs in the first 'yum install' stanza. This of course drags in a minimal base system.
   a) Size: 333 MB
2. Then the script does a yum group-install of core and development.
   a) Size: 517 MB
3. The next stanza installs a list of packages that we always install on workers in the Tier-1. (This could be skipped I think, relying instead on the HEP_Oslibs_SL6 RPM to pull in dependencies)
   a) Size: 1.1 GB
4. Then we install the UMD-3 middleware.
   a) Size: 1.7 GB
5. Finally we install HEP_Oslibs_SL6 and condor
   a) Size: 1.8 GB

6.2 Final image setup

After the software is installed the script expunges any remaining space-gobbling features like man pages, language files, icons, etc. and then the size falls to **1.5 GB**.

At the end of the script we add local customization – the group and user configuration is appended to passwd, shadow, group, gshadow in the target etc directory; the Tier-1 profile is added as well as the local condor configuration. A tarball is created, and the final size of this file is **506 MB**.

To create a truly portable image for use outside of the Tier-1, we could use release RPMs from global open HTTP mirrors instead.

The final 'local' configuration could also be done on the container at launch time, or could be applied to a separate layered image, so that the generic image would be untainted.

As part of the image creation we have installed a locally-built RPM named dumb-init. The ***dumb-init***[1] application is useful in container environments because it allows better control over launch-time commands. See ahead in the Container section on how it is used.

7 Importing Images to the Registry

The syntax used for importing images into a registry is:

```
Usage:  docker import [OPTIONS] file|URL|- [REPOSITORY[:TAG]]
```

We import the tarball and then 'tag' it with a repository group and tag name:

```
# docker import eight-sl6.tar.gz  sl6:umd3condv4
sha256:762758303a84d2d8eb3ad690b2807ced68f7302c268335b33929ce0aadda0773

# docker images  ### it is imported but not yet tagged:
REPOSITORY                                TAG        IMAGE ID     CREATED            SIZE
sl6                                       umd3condv4 762758303a84 21 seconds ago     857.6 MB
pps04.lcg.triumf.ca:5000/sl6-umd3condv3 latest     bb7c8d288fee 3 days ago         1.445 GB
sl6                                       umd3condv3 bb7c8d288fee 3 days ago         1.445 GB
pps04.lcg.triumf.ca:5000/sl6-umd3condv2 latest     7d49ace6c693 6 days ago         1.391 GB
. . .

# docker tag sl6:umd3condv4 pps04.lcg.triumf.ca:5000/sl6-umd3condv4
# docker images
REPOSITORY                                TAG        IMAGE ID     CREATED            SIZE
pps04.lcg.triumf.ca:5000/sl6-umd3condv4 latest     762758303a84 4 minutes ago      857.6 MB
sl6                                       umd3condv4 762758303a84 4 minutes ago      857.6 MB
pps04.lcg.triumf.ca:5000/sl6-umd3condv3 latest     bb7c8d288fee 3 days ago         1.445 GB
sl6                                       umd3condv3 bb7c8d288fee 3 days ago         1.445 GB
pps04.lcg.triumf.ca:5000/sl6-umd3condv2 latest     7d49ace6c693 6 days ago         1.391 GB
. . .
```

Finally we push it officially so that it is seen by clients:

```
# docker push pps04.lcg.triumf.ca:5000/sl6-umd3condv4
The push refers to a repository [pps04.lcg.triumf.ca:5000/sl6-umd3condv4]
2d4dc4e96366: Preparing
2d4dc4e96366: Pushing [======>                                            ] 110.1 MB/857.6 MB
. . .
2d4dc4e96366: Image successfully pushed
Pushing tag for rev [762758303a84] on
 {http://pps04.lcg.triumf.ca:5000/v1/repositories/sl6-umd3condv4/tags/latest}
```

Now clients can see it:

1 https://github.com/Yelp/dumb-init

```
[root@wn024 ~]# docker search sl6
INDEX       NAME                                                DESCRIPTION  STARS OFFICIAL AUTOMATED
triumf.ca   pps04.lcg.triumf.ca:5000/library/sl6-first                       0
triumf.ca   pps04.lcg.triumf.ca:5000/library/sl6-new                         0
triumf.ca   pps04.lcg.triumf.ca:5000/library/sl6-umd3                        0
triumf.ca   pps04.lcg.triumf.ca:5000/library/sl6-umd3cond                    0
triumf.ca   pps04.lcg.triumf.ca:5000/library/sl6-umd3condv2                  0
triumf.ca   pps04.lcg.triumf.ca:5000/library/sl6-umd3condv3                  0
triumf.ca   pps04.lcg.triumf.ca:5000/library/sl6-umd3condv4                  0
```

8 Docker Containers

When you launch a Docker container, you need to pull an image if it is not yet local. The docker command knows where to find images by virtue of the configuration as outlined in section 4 above. The image will be pulled automatically – here is an example:

```
# docker run -d -it --cap-add SYS_ADMIN -v /home:/home \
 /etc/grid-security:/etc/grid-security -v /cvmfs:/cvmfs \
 /opt/glite:/opt/glite -v /etc/localtime:/etc/localtime \
 -p 4080:4080 sl6-umd3condv4  /root/init


Unable to find image 'sl6-umd3condv4:latest' locally
Trying to pull repository pps04.lcg.triumf.ca:5000/sl6-umd3condv4 ...
Pulling repository pps04.lcg.triumf.ca:5000/sl6-umd3condv4

762758303a84: Extracting [============>                          ] 72.42 MB/312.8 MB
. . .
Status: Downloaded newer image for pps04.lcg.triumf.ca:5000/sl6-umd3condv4:latest
pps04.lcg.triumf.ca:5000/sl6-umd3condv4: this image was pulled from a legacy
registry.  Important: This registry version will not be supported in future versions
of docker.
```

Note the warning message above, concerning the registry version. Our future work on Docker image registries needs to focus on Docker Distribution instead because of this issue. Indeed, at Ansible version 1.9 any Ansible docker command failed with this warning. At least at Ansible 1.10 a better supported 'docker_container' command succeeds and issues the warning.

8.1 Running Docker containers

8.1.1 Bind mounts

A very useful feature of docker containers is the ability to bind-mount volumes. We make liberal use of this in early docker worker node containers to preserve disk space and to reuse components. The list of bind-mounts used in the test bed:

- /etc/grid-security
  → let the Engine host manage CRLs
- /etc/localtime
  → set the timezone in the container to match the Engine host
- /opt/glite
  → probably will not be needed, but this directory on the Engine host contains yaim configuration settings that we could access if needed
- /cvmfs
  → we want to avoid nfs mounts in the container. Instead we install cvmfs on the Engine host, and bind-mount it on the container.
  → note that I did configure the Engine host to never unmount the cvmfs trees – we need to test to see if this is really necessary:

```
[root@wn024 ~]# tail -1 /etc/auto.master
/cvmfs   /etc/auto.cvmfs --timeout 0
```

- /home
  → this is the traditional scratch space for user jobs in the Tier-1. Therefore the Engine host had a large home directory with user and group ID numbers that must match the uid/gid setup on the docker container.
  → The Engine host can be responsible for cleanup of scratch space and home directories via its own cron jobs. At the Tier-1, the disk-checking cron jobs would also continue to run from the Engine host – however, assuming a condor configuration the way of stopping the container's condor_master in case of imminent disk failure would need to be revisited.

8.1.2 Port Mappings

To simplify the configuration we start condor using the shared-port option:

```
# diff condor_config.local condor_config.local.t1-ppsce
150,154d149
<
< ## Tier-1 docker test settings
< DISCARD_SESSION_KEYRING_ON_STARTUP = False
< USE_SHARED_PORT = True
< SHARED_PORT_ARGS = -p 4080
```

We can then map the internal port to any unused port on the Engine host with the command-line option '-p'; e.g.
-p 4080:4080

or

-p 4088:4080

8.1.3 Hostnames and IP addresses

Docker networking is not like virtualization networking – unless you create user-defined subnets then you cannot control the IP address of containers at launch – Docker Engine will sequentially assign IP addresses from its IP base. However to set the local hostname of a container to the DNS-assigned hostname one can use the trick of looking up the hostname from its IP address in the launch script and assigning it. However, the SYS_ADMIN capability is needed to allow the container to change its own hostname. This is less secure – we need to look more carefully at this issue – see the link in the footnote[1].

```
# cat /root/init
#!/usr/bin/dumb-init /bin/bash
## get the ip address and hostname - set the hostname before starting condor
thisip=$(ip -4 route get 1 | awk '{print $NF;exit}')
if [ "$thisip" != "" ] ; then
  thishost=$(host $thisip) 2>/dev/null
  if [ "$thishost" != "" ] ; then
    h=$(echo "${thishost##* }")
    h=$(echo "${h%?}")
    hostname $h
  fi
fi
## set up condor environment before running condor_master in the foreground
. /etc/sysconfig/condor
/usr/sbin/condor_master -f
```

8.2 Docker and Ansible

The command-line is a bit unwieldy when launching containers. This is a use-case for tools like Ansible. We have an example docker role named 'docker-worker' which sets up a docker environment on an SL7 worker node. Then we can use an Ansible playbook to launch containers; here is the current example:

```
# ansible-playbook docker/condor-container.yml \
  -e "target='wn024' thisname=vn0317 external_port=4081 image=sl6-umd3condv4"
```

1 https://docs.docker.com/engine/reference/run/#/runtime-privilege-and-linux-capabilities

```
# cat playbooks/docker/condor-container.yml
---
# Usage:
#  ansible-playbook THIS_FILE -e "image='sl6' name='somename'"
#  the internal and external port assignments can be given default values or
#  can be overridden.  We should make the command into a variable as well.

- name: Launch docker worker condor-based containers
  hosts: "{{ target }}"
  gather_facts: false
  user: root

  tasks:
  - name: Only run if this is a docker server
    fail: msg="This is not a docker server"
    when: not is_docker_server

  - name: Launch a test condor-enabled htcondor container
    docker_container:
      name: "{{ thisname }}"
      image: "{{ image }}"
      exposed_ports: "{{ port }}"
      published_ports: "{{ external_port }}:{{ port }}"
      command: "/root/init"
      capabilities: SYS_ADMIN
      volumes:
          - /etc/grid-security
          - /etc/localtime
          - /cvmfs
          - /home
          - /opt/glite
```

9 Next Steps

So far we have only used containers to run OPS jobs in a condor environment in the pre-production setup at the Tier-1. Aside from getting experience with running containers for ATLAS jobs, we also need look at container trust issues, maintenance and logging.

9.1 Distribution host

The next step for a Docker Registry is to move to Docker Distribution. At this point we begin to look at securely signing images, as well as securely transporting images.

Here are a few important links to read concerning Docker security:

- https://blog.docker.com/2013/11/introducing-trusted-builds/
- https://access.redhat.com/blogs/766093/posts/1976473
- https://titanous.com/posts/docker-insecurity

9.2 Logging

We will want to capture logs from containers in a production environment. There are a number of logging options[1] that need to be investigated. Another option is to create container-named sub-directories in /var/log/ on the Engine host and bind-mount those volumes to the container – for example – capturing the container's internal logs from /var/log/condor under a sub-directory on the Docker Engine host at /var/log/SOME_NAME_timestamp/ or elsewhere as needed. This is easily done with Ansible at container launch time.

9.3 Image Maintenance and Tracking

We will need a plan on how images are maintained and tracked:

- When and where do we patch the images? Should they be patched at the registry hub, or would you update them inside the container before launching the application?
- How will we track changes? Should images, or just their scripts, and/or their signature hashes be tracked in a version control system?
- What monitoring tools do we need?

10 Vocabulary

There are a number of technologies associated with Docker that are not mentioned in this document. In version 2 of this document I will provide a short reference of terms you may come across in your research, with a short definition of each.

11 Appendix

11.1 Docker Registry YAML initial configuration

```
# All other flavors inherit the `common' config snippet
common: &common
    issue: '"docker-registry server"'
    # Default log level is info
    loglevel: _env:LOGLEVEL:info
    # Enable debugging (additional informations in the output of the _ping endpoint)
    debug: _env:DEBUG:false
    # By default, the registry acts standalone (eg: doesn't query the index)
    standalone: _env:STANDALONE:true
    # The default endpoint to use (if NOT standalone) is index.docker.io
    #index_endpoint: _env:INDEX_ENDPOINT:https://index.docker.io
    # Storage redirect is disabled
    storage_redirect: _env:STORAGE_REDIRECT
```

1 https://docs.docker.com/engine/admin/logging/overview/

```
    # Token auth is enabled (if NOT standalone)
    disable_token_auth: _env:DISABLE_TOKEN_AUTH
    # No priv key
    privileged_key: _env:PRIVILEGED_KEY
    # No search backend
    search_backend: _env:SEARCH_BACKEND:sqlalchemy
    # SQLite search backend
    sqlalchemy_index_database: _env:SQLALCHEMY_INDEX_DATABASE:sqlite:////data/docker/index/docker-registry.db

    #    # Mirroring is not enabled
    #    mirroring:
    #        source: _env:MIRROR_SOURCE # https://registry-1.docker.io
    #        source_index: _env:MIRROR_SOURCE_INDEX # https://index.docker.io
    #        tags_cache_ttl: _env:MIRROR_TAGS_CACHE_TTL:172800 # seconds

    cache:
        host: _env:CACHE_REDIS_HOST
        port: _env:CACHE_REDIS_PORT
        db: _env:CACHE_REDIS_DB:0
        password: _env:CACHE_REDIS_PASSWORD

    # Enabling LRU cache for small files
    # This speeds up read/write on small files
    # when using a remote storage backend (like S3).
    cache_lru:
        host: _env:CACHE_LRU_REDIS_HOST
        port: _env:CACHE_LRU_REDIS_PORT
        db: _env:CACHE_LRU_REDIS_DB:0
        password: _env:CACHE_LRU_REDIS_PASSWORD

    #    # Enabling these options makes the Registry send an email on each code Exception
    #    email_exceptions:
    #        smtp_host: _env:SMTP_HOST
    #        smtp_port: _env:SMTP_PORT:25
    #        smtp_login: _env:SMTP_LOGIN
    #        smtp_password: _env:SMTP_PASSWORD
    #        smtp_secure: _env:SMTP_SECURE:false
    #        from_addr: _env:SMTP_FROM_ADDR:docker-registry@localdomain.local
    #        to_addr: _env:SMTP_TO_ADDR:noise+dockerregistry@localdomain.local

    # Enable bugsnag (set the API key)
    bugsnag: _env:BUGSNAG

local: &local
    <<: *common
    storage: local
    storage_path: _env:STORAGE_PATH:/data/docker/registry

# This is the default configuration when no flavor is specified
dev: &dev
    <<: *local
    loglevel: _env:LOGLEVEL:debug
    debug: _env:DEBUG:true
    search_backend: _env:SEARCH_BACKEND:sqlalchemy

## To specify another flavor, set the environment variable SETTINGS_FLAVOR
## $ export SETTINGS_FLAVOR=prod
#prod:
#    <<: *s3
#    storage_path: _env:STORAGE_PATH:/prod
```