Best Practices

Instructions

1. Review each guideline
2. Add your comments and suggestions to the "Comments" column
3. The disposition will be determined based on of your input and discussion with the team.
4. Once you have input your comments, close the JIRA task to signal the team that you have provided your input.

Scope

The guidelines listed here:

- Are intended to address the build and validation processes.
- Don't address the fundamentals or principles of container images.

General Comments:

- add guideline addressing base images, e.g. example in project proposal re: alpine base image (FS)
- add guideline addressing multi-platform images (FS)
- add guideline addressing image names, e.g. "db" discouraged, "onap-component-db" preferred, e.g. "music-db" (FS)
- add guideline addressing proper use of onap image repo (FS)

General Guidelines	Comments	Disposition
1. **Understand build context** When executing "`docker build`", the current working directory is the *build context*. By default, the Dockerfile is assumed to be in the current working directory. Irrespective of where the `Dockerfile` is located, all recursive contents of files and directories in the current directory are sent to the Docker daemon as the build context. Inadvertently including files that are not necessary for building an image results in a larger build context and larger image size. This can increase the time to build the image, time to pull and push it, and the container runtime size.	"this can increase'..." I suggest the very first guideline should be a general statement about image size, as many of these individual items address that general concern	
2. Exclude with .dockerignore Exclude files not relevant to the build with a `.dockerignore` file. This file supports exclusion patterns similar to `.gitignore` files.		
3.Use multi-stage builds Multi-stage builds reduces the size of an image, without worrying about the number of intermediate layers and files. An image is built during the final stage of the build process. The number of image layers can be minimized by leveraging build cache. For a build that contains several layers, order them from the less frequently changed (re-use build cache) to the more frequently changed: • Install tools you need to build your application • Install or update library dependencies • Generate your application	"the number of layers..." Perhaps say "see below 'Re-use the build cache'	
4. Don’t install unnecessary packages Avoid installing extra or unnecessary packages. This will reduce complexity, dependencies, file sizes, and build times, Don't include a text editor in a database image.		

5. Decouple applications Apply the principle of "separation of concerns." Each container should have only one concern. Decoupling applications into multiple containers makes it easier to reuse containers.		
6. Minimize the number of layers The instructions `RUN`, `COPY`, `ADD` create layers and directly increase the size of the build. Use multi-stage builds, to only copy the artifacts you need into the final image. Tools and debug information can be added to intermediate build stages without increasing the size of the final image.	"tools and debug info..." I don't understand this. Perhaps an example would be helpful	
7. Sort multi-line arguments To minimize duplication of packages and make the list of packages much easier to update, sort multi-line arguments alphanumerically.		
8. Re-use the build cache As each instruction in the Dockerfile is examined, the builder looks for an existing image in its cache that can be reused, rather than creating a duplicate image. • For the `ADD` and `COPY` instructions, the contents of the file(s) in the image are examined and a checksum is calculated for each file. The last-modified and last-accessed times of the file(s) are not considered in these checksums. During the cache lookup, the checksum is compared against the checksum in the existing images. If anything has changed in the file(s), such as the contents and metadata, then the cache is invalidated. • Aside from the `ADD` and `COPY` commands, cache checking does not look at the files in the container to determine a cache match. • For example, when processing a `RUN apt-get -y update` command the files updated in the container are not examined to determine if a cache hit exists. In that case just the command string itself is used to find a match. Once the cache is invalidated, all subsequent `Dockerfile` commands generate new images and the cache is not used.		
Build File Instructions	**Comments**	**Disposition**
FROM Use current official repositories for base images. ONAP images must be vendor agnostic; ensure that the base images are cpu architecture-agnostic.	are there base images that are multi-pltfrom, or is there a need to create multiple images for multiple targets? eg. there is 'alpine' and 'arm64v8/alpine'	
LABEL Labels are unique key-value pairs used to add metadata to container images and containers. They help organize images by project, add licensing information, or to support build and CI pipeline. An image can have more that one label. For each label, begin a new line with "LABEL" and add one or more key-value pairs.		

RUN

To make the build file (e.g. Dockerfile) more readable, understandable and maintainable:

RUN apt-get

1. Split a long or complex statement into multiple lines
2. Separate each line by a backslash (\)
3. Avoid RUN apt-get upgrade and RUN apt-get dist-upgrade.
    a. Some packages from the parent image may not upgrade in a container.
4. If you must update a package (e.g. bar) use "apt-get install -y bar"
5. Install the latest package versions with no further coding or manual intervention by combining "RUN apt-get update" and "apt-get install -y" in a single RUN statement:
    a. RUN apt-get update && apt-get install -y
6. Use "version pinning." It will:
    a. Force the build to use a particular package version regardless of what's in the cache
    b. Reduce failures due to unexpected changes in required packages

Well-formatted example:

```
RUN apt-get update && apt-get install -y \
    aufs-tools \
    automake \
    build-essential \
    curl \
    dpkg-sig \
    libcap-dev \
    libsqlite3-dev \
    mercurial \
    reprepro \
    ruby1.9.1 \
    ruby1.9.1-dev \
    s3cmd=1.1.* \
 && rm -rf /var/lib/apt/lists/*
```

ERRORS IN STAGES OF A PIPE

7. Use "set -o pipefail &&" to ensure that the RUN command only succeeds if all stages of a pipe succeed.

```
RUN ["/bin/bash", "-c", "set -o pipefail && wget -O - https://myapp.net
| wc -l >= b"]
```

CMD

This instruction provides defaults to run the application packaged in a container image. It should always be used in the form:

```
CMD [“executable”, “arg1”, “arg2”…]
```

Typically, CMD should run an interactive shell. That way users get a usable shell when they execute "docker run -it ..." For example:

CMD ["sh", "-c", "echo $ENV"]

```
CMD ["python"]
```

```
CMD [“php”, “-a”]
```

Note: If the user provides arguments to "docker run", they override the defaults specified in "CMD."

EXPOSE

Use well-known ports for your application. For example, an image containing Apache web server should use `EXPOSE 80`. An image containing MongoDB should use `EXPOSE 27017` and so on and so forth.

ENV

Use ENV to avoid hard-coding values for variables and parameters in the your build file. ENV can parameterize container variables. For example, the version of the software in the container (VERSION), the PATH environment variable and other execution environment variables (MAJOR).

```
ENV MAJOR 1.3
ENV VERSION 1.3.4
RUN curl -SL http://example.com/postgres-$VERSION.tar.xz | tar -xJC /usr
/src/postgress && …
ENV PATH /usr/local/postgres-$MAJOR/bin:$PATH
```

Each `ENV` command creates a new intermediate layer. Even if you unset the environment variable in a future layer, it still persists in this layer. Use a `RUN` command with shell commands, to set, use, and unset the variable all in a single layer. Separate your commands with `&&`.

Example

```
RUN export ADMIN_USER="seneca" \
    && echo $ADMIN_USER > ./seneca \
    && unset ADMIN_USER
```

ADD or COPY

`ADD and COPY are functionally similar but COPY` is preferred because `it` only supports copying of local files into the container and it's more transparent than `ADD`.

ADD is recommended for local tar file auto-extraction into the image (e.g. `ADD rootfs.tar.xz /` .).

If you have to copy several files from your context, `COPY` them individually, instead of all at once. That way each step is only re-execute if the specifically required files change.

To reduce the number of layers and the image size, don't use `ADD` to download packages from URLs. Use `curl` or `wget and` delete the files you no longer need after they've been extracted.

Example:

```
RUN mkdir -p /usr/src/ether \
    && curl -SL http://vacuum.com/huge.tar.xz \
    | tar -xJC /usr/src/ether \
    && make -C /usr/src/ether all
```

ENTRYPOINT

Use `ENTRYPOINT` to set the image's main command. That allows the image to be executed as though it was that command. Use `CMD` to set the default arguments.

Example:

By setting

ENTRYPOINT ["my_command"]

CMD["–version"]

The image can be run as

$ docker run my_command

or

$ docker rum my_command --verbose -n 10

`ENTRYPOINT` can also be used in combination with a script when more than one execution step is required.

Copy the script into the container and run it via `ENTRYPOINT` on container start.

Example:

```
COPY ./docker-entrypoint.sh /
ENTRYPOINT ["/docker-entrypoint.sh"]
CMD ["redis"]
```

User can then execute:

$ docker run redis

VOLUME

The `VOLUME` instruction should be used for any mutable or user-serviceable parts of the image.

VOLUME exposes database storage areas, configuration storage, or files/folders created by the container.

USER

Do not run containers as root. Use USER to change to an non-root user.

Create the user and group as in this example:

```
RUN groupadd -r postgres && useradd --no-log-init -r -g postgres postgres
```

.

Avoid installing or using `sudo`. If you need to, use "gosu" instead.

To minimize the number of layers, avoid switching `USER` back and forth frequently.

WORKDIR

Always use absolute paths for your `WORKDIR`.

Use `WORKDIR` instead of cd commands like "`RUN cd … && do-something`." They could be hard to read, troubleshoot, and maintain.

ONBUILD `ONBUILD` executes on children images derived FROM the current image. ONBUILD can be seen as an instruction the parent `Dockerfile` gives to the child `Dockerfile`. Images built from `ONBUILD` should get a separate tag, for example: `java:8-onbuild` or `java:9-onbuild`.		

Build Environment

There are concerns in building images outside the scope of the Dockerfile itself

See here for image tagging guidelines: Independent Versioning and Release Process#StandardizedDockerTagging

The CIA has undertaken to reduce image size and ensure images may be build and run on multiple platforms.

This activity has exposed the team to differing approaches taken by project teams to build images. This page presents best practices for building images, as an extension the the best practices currently described for Dockerfiles (add link)

Note: the best practices are currently documented both
here https://wiki.onap.org/display/DW/Best+Practices
and here https://wiki.onap.org/display/DW/Container+Image+Minimization+Guidelines

There are two scenarios for image builds; Local and CSIT.

Local build/test

Local builds are required in order to validate assets such as poms, Dockerfiles, and scripts.
Developers should not push assets into a project repo until there have been validated, and the CSIT pipeline requires that assets are in repos, ergo, local builds are mandatory.

Local builds involve not simply building images, but running them and testing wether they are behaving properly. These steps are often not documented, and in some cases require reverse-engineering to get working.

Documentation

Every project should document steps to building images and testing them locally. This should be part of the project ReadTheDocs. The CIA team can contribute to documentation as it works through projects. Documentation should be addressed to an audience that has general Unix and programming skills, but does not have knowledge of project or arcane topics such as complex maven definitions.

Recommendation: project RTD includes materail describing local build and test of docker images

maven

Many projects build images as part of the workflow of building application targets. While this provides some convenience, there are drawbacks.

This approach tightly-couples the application build process with the container packaging process, perhaps unnecessarily so.

Other projects provide shell scripts to build images. This has the benefit of separating the build and packaging steps

Recommendation: TBD

docker-maven-plugin

There are currently several maven plugins in use by ONAP project teams.

The spotify docker-maven-plugin is no longer recommended, as Spotify encourages the use of pre-written Dockerfiles, as opposed to generating a Dockerfile via plugin configuration.
See https://github.com/spotify/docker-maven-plugin

In addition, this plugin does not execute on all platforms, including Arm, and therefore should not be used.
See https://github.com/spotify/docker-maven-plugin/issues/233

The spotify dockerfile-maven-plugin has the benefit of supporting a pre-written Dockerfile, but also has issues running on all platforms, and therefore should not be used (TBD - confirm details).

Finally, the fabric8 docker-maven-plugin has been used successfully to build images, using explicit Dockerfiles, on multiple platforms.

Recommendation: If one uses maven to build images, use the fabric8 docker-maven-plugin (v 0.28.0, as previous don't work on Arm)

Pull Repo

For projects that employ maven to build images, care must be taken to ensure that images may be built on a number of platforms (e.g. amd64, arm64, etc)

Normative base images are available on DockerHub as multi-platfrom FAT manifests (i.e. a universal label that lists platform specific images). However, projects often make use of the ONAP nexus registry as a source of images. While this serves as an image cache that improves CSIT build performance, it breaks multi-platfrom compatibility as only a platform-specific image is cached, not the multi-platform manifest.

ONAP provides a maven settings.xml file which defines the following:
<docker.pull.registry>nexus3.onap.org:10001</docker.pull.registry>

This in effect hardcodes an amd64 base image for a project using maven as the image build tool.

To address this issue, and avoid changing the ONAP provided settings.xml, one may use the following directive in the configuration section of the fabric8 docker-maven-plugin
<pullRegistry>${docker.pull.registry}</pullRegistry>

and set the docker.pull.registry variable when invoking maven, e.g.
> mvn install -Pdocker -Ddocker.pull.registry=docker.io

Push Repo

In order to test the maven lifecycle to push images, a developer must instantiate a local docker registry, configure the pom appropriately, and invoke mvn appropriately.

Start a local docker image repo:
see: https://docs.docker.com/registry/deploying/
> docker run -d -p 5000:5000 --restart=always --name registry registry:2

Add the following directive in the configuration section of the fabric8 docker-maven-plugin
<pushRegistry>${docker.push.registry}</pushRegistry>

and set the docker.push.registry variable when invoking maven, e.g.
> mvn install -Pdocker -Ddocker.push.registry=localhost:5000 docker:push
Note invocation of docker:deploy lifecycle

Notes:

1) alpine images use sh rather then bash. Some side-effects
- In Dockerfile, CMD and ENTRYPOINT
bash - ENTRYPOINT ["startup.sh"]
sh - ENTRYPOINT ["sh", "startup.sh"]