Build Images with Advanced Containerfile Instructions

Objectives

- Create a Containerfile that uses best practices.

Advanced Containerfile Instructions

You can package your application in a container with the application runtime dependencies. However, developers commonly create containers that have a number of downsides, for example:

- The container is bound to a specific environment, and requires a rebuild before deploying to a production environment.
- The container contains developer tools, such as a debugger, text editors, or compilers.
- The container contains build-time dependencies that are not necessary at runtime.
- The container generates a large volume of files stored on the copy-on-write file system, which limits the performance of the application.

This section focuses on advanced container patterns that help you limit previously mentioned issues, such as:

- Reducing image storage footprint by using the multistage container build pattern.
- Customizing container runtime with environment variables.
- Using volumes to decrease the container size and increase the performance of writing files.

The ENV Instruction

The `ENV` instruction lets you specify environment dependent configuration, for example, hostnames, ports or usernames. The containerized application can use the environment variables at runtime.

To include an environment variable, use the `key=value` format. The following example declares a `DB_HOST` variable with the database hostname.

```
ENV DB_HOST="database.example.com"
```

Then you can retrieve the environment variable in your application. The following example is a Python script that retrieves the `DB_HOST` environment variable.

```
from os import environ

DB_HOST = environ.get('DB_HOST')

# Connect to the database at DB_HOST...
```

The ARG Instruction

Use the ARG instruction to define build-time variables, typically to make a customizable container build.

You can optionally configure a default build-time variable value if the developer does not provide it at build time. Use the following syntax to define a build-time variable:

```
ARG key[=default value]
```

When you build the container image, use the `--build-arg` flag to set the value, such as `podman build --build-arg key=example-value` for the preceding example.

Consider the following Containerfile example:

```
ARG VERSION="1.16.8" \
    BIN_DIR=/usr/local/bin/

RUN curl "https://dl.example.io/${VERSION}/example-linux-amd64" \
        -o ${BIN_DIR}/example
```

If you do not provide the `--build-arg` flag, then Podman uses the default values during the build process. However, you can change the values during the build process without changing the Containerfile by using the ARG instruction.

Developers commonly configure the ENV instructions by using the ARG instruction. This is useful for preserving the build-time variables for runtime.

Consider the following Containerfile example:

```
ARG VERSION="1.16.8" \
    BIN_DIR=/usr/local/bin/

ENV VERSION=${VERSION} \
    BIN_DIR=${BIN_DIR}

RUN curl "https://dl.example.io/${VERSION}/example-linux-amd64" \
        -o ${BIN_DIR}/example
```

In the preceding example, the ENV instruction uses the value of the ARG instruction to configure the run-time environment variables.

If you do not configure the ARG default values but you must configure the ENV default values, then use the `${VARIABLE:-DEFAULT_VALUE}` syntax, such as:

```
ARG VERSION \
    BIN_DIR

ENV VERSION=${VERSION:-1.16.8} \
    BIN_DIR=${BIN_DIR:-/usr/local/bin/}

RUN curl "https://dl.example.io/${VERSION}/example-linux-amd64" \
        -o ${BIN_DIR}/example
```

The VOLUME Instruction

Use the `VOLUME` instruction to persistently store data. The value is the path where Podman mounts a persistent volume inside of the container. The `VOLUME` instruction accepts more than one path to create multiple volumes.

For example, the following Containerfile creates a volume to store PostgreSQL data.

```
FROM registry.redhat.io/rhel9/postgresql-13:1

VOLUME /var/lib/pgsql/data
```

In the previous Containerfile, if you add instructions that update `/var/lib/pgsql/data` after the VOLUME instruction, then those instructions are ignored.

To retrieve the local directory used by a volume, you can use the `podman inspect` *`VOLUME_ID`* command.

```
[user@host ~]$ podman inspect VOLUME_ID
[
    {
        "Name": "VOLUME_ID",
        "Driver": "local",
        "Mountpoint": "/home/your-
name/.local/share/containers/storage/volumes/VOLUME_ID/_data",
        ...output omitted...
        "Anonymous": true
    }
]
```

The `Mountpoint` field gives you the absolute path to the directory where the volume exists on your host file system. Volumes created from the `VOLUME` instruction have a random ID in the `Name` field and are considered `Anonymous` volumes.

To remove unused volumes, use the `podman volume prune` command.

```
[user@host ~]$ podman volume prune
WARNING! This will remove all volumes not used by at least one container. The following
volumes will be removed:
8c0c...bcb8c
Are you sure you want to continue? [y/N] y
8c0c...bcb8c
```

You can create a *named* volume by using the `podman volume create` command.

```
[user@host ~]$ podman volume create VOLUME_NAME
VOLUME_NAME
```

You can also format the `podman volume ls` command to include the `Mountpoint` field of every volume. Persistent data storage with volumes is covered in depth later in the course.

```
[user@host ~]$ podman volume ls \
 --format="{{.Name}}\t{{.Mountpoint}}"
0a8c...82c2  /home/your-name/.local/share/containers/storage/volumes/0a8c...82c2/_data
252d...b2ed  /home/your-name/.local/share/containers/storage/volumes/252d...b2ed/_data
```

The ENTRYPOINT and CMD Instructions

The `ENTRYPOINT` and `CMD` instructions specify the command to execute when the container starts. A valid Containerfile must have at least one of these instructions.

The `ENTRYPOINT` instruction defines an executable, or command, that is always part of the container execution. This means that additional arguments are passed to the provided command.

Consider the following example:

```
FROM registry.access.redhat.com/ubi8/ubi-minimal:8.5
ENTRYPOINT ["echo", "Hello"]
```

Running a container from the previous Containerfile with no arguments prints "Hello".

```
[user@host ~]$ podman run my-image
Hello
```

If you provide the container with the "Red" and "Hat" arguments, then it prints "Hello Red Hat".

```
[user@host ~]$ podman run my-image Red Hat
Hello Red Hat
```

If you only use the `CMD` instruction, then passing arguments to the container overrides the command provided in the `CMD` instruction.

```
FROM registry.access.redhat.com/ubi8/ubi-minimal:8.5
CMD ["echo", "Hello", "Red Hat"]
```

Running a container from the previous Containerfile with no arguments prints `Hello Red Hat`.

```
[user@host ~]$ podman run my-image
Hello Red Hat
```

Running a container with the argument `whoami` overrides the `echo` command.

```
[user@host ~]$ podman run my-image whoami
root
```

When a Containerfile specifies both `ENTRYPOINT` and `CMD` then `CMD` changes its behavior. In this case the values provided to `CMD` are passed as default arguments to the `ENTRYPOINT`.

```
FROM registry.access.redhat.com/ubi8/ubi-minimal:8.5
ENTRYPOINT ["echo", "Hello"]
CMD ["Red", "Hat"]
```

The previous example prints `Hello Red Hat` when you run the container.

```
[user@host ~]$ podman run my-image
Hello Red Hat
```

If you provide the argument `Podman` to the container, then it prints `Hello Podman`.

```
[user@host ~]$ podman run my-image Podman
Hello Podman
```

The `ENTRYPOINT` and `CMD` instructions have two formats for executing commands:

Text array
The executable takes the form of a text array, such as:

```
ENTRYPOINT ["executable", "param1", ... "paramN"]
```

In this form you must provide the full path to the executable.

String form
The command and parameters are written in a text form, such as:

```
CMD executable param1 ... paramN
```

The string form wraps the executable in a shell command such as `sh -c "executable param1 ... paramN"`. This is useful when you require shell processing, for example for variable substitution.

You might need to change the container entrypoint at runtime, for example for troubleshooting. Consider the following Containerfile:

```
FROM registry.access.redhat.com/ubi8/ubi-minimal:8.5

LABEL GREETING="World"

ENTRYPOINT echo Hello "${GREETING}"
```

When you run the container, you notice unexpected output:

```
[user@host ~]$ podman run my-image
Hello
```

You can execute the `env` command to print the environment variables in the container. However, the preceding container uses the `ENTRYPOINT` instruction, which means that you cannot change the `echo` command by adding an argument to the container execution:

```
[user@host ~]$ podman run my-image env
Hello
```

In that case, you can overwrite the entrypoint by using the `podman run --entrypoint` command.

```
[user@host ~]$ podman run --entrypoint env my-image
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
TERM=xterm
container=oci
HOME=/root
HOSTNAME=ba00a7663a93
```

Consequently, you verify that the `GREETING` environment variable is not set. This is because the developer that created the Containerfile used the `LABEL` instruction instead of the `ENV` instruction.

Podman Secrets

A secret is a blob of sensitive information required by containers at runtime. This can be usernames, passwords, or keys. For example, if your containerized application requires credentials for connecting to a database, then you must store those credentials as a secret. After creating the secret, you must instruct the application container to make the credentials available to the application when it starts. To manage this process, use the `podman secret` subcommands.

Creating Podman Secrets

With Podman, you can create secrets by using either a file, or by passing the sensitive information to the standard input (STDIN). To create a secret from a file, run the `podman secret create` subcommand specifying the name of the file containing the sensitive

information, and the name of the secret to create as arguments. The output of the `podman secret create` subcommand displays the secret ID.

```
[user@host ~]$ echo "R3d4ht123" > dbsecretfile
[user@host ~]$ podman secret create dbsecret dbsecretfile
9c2400836ee16ed07d86a3122
```

Run the `podman secret create` subcommand specifying the secret name and `-` as arguments. The `-` argument instructs podman to read the sensitive information from standard input.

```
[user@host ~]$ printf "R3d4ht123" | podman secret create dbsecret2 -
875a1e46fa64639756968c644
```

The `podman secret create` command supports different drivers to store the secrets. You can use the `--driver` or `-d` option to specify one of the supported secret drivers:

`file` (default)
Stores the secret in a read-protected file

`pass`
Stores the secret in a GPG-encrypted file.

`shell`
Manages the secret storage by using a custom script.

To list stored secrets, use the `podman secret ls` subcommand.

```
[user@host ~]$ podman secret ls
ID                         NAME         DRIVER      CREATED             UPDATED
875a1e46fa64639756968c644  dbsecret2    file        About a minute ago  About a minute
ago
9c2400836ee16ed07d86a3122  dbsecret     file        14 minutes ago      14 minutes
```

When you no longer need a secret, you can remove it by running the `podman secret rm` command, and providing the name of the secret as argument.

```
[user@host ~]$ podman secret rm dbsecret2
875a1e46fa64639756968c644
```

Running Containers With Podman Secrets

To make secrets available for use to a container, execute the `podman run` command with the `--secret` option, and specify the name of the secret as parameter. You can use the `--secret` option multiple times for multiple secrets.

```
[user@host ~]$ podman run -it --secret dbsecret \
--name myapp registry.access.redhat.com/ubi8/ubi /bin/bash
[root@f9fedddbc81a /]# cat /run/secrets/dbsecret
R3d4ht123
```

When you use secrets, Podman retrieves the secret and places it on a tmpfs volume and then mounts the volume inside the container in the `/run/secret` directory as a file based on the name of the secret.

To prevent secrets from being stored in an image, neither the `podman commit` nor `podman export` commands copy the secret data to an image or .tar file.

Multistage Builds

A multistage build uses multiple `FROM` instructions to create multiple independent container build processes, also called *stages*. Every stage can use a different base image and you can copy files between stages. The resulting container image consists of the last stage.

Multistage builds can reduce the image size by only keeping the necessary runtime dependencies. For example, consider the following example, where a Node application is containerized.

```
FROM registry.redhat.io/ubi8/nodejs-14:1

WORKDIR /app
COPY . .

RUN npm install
RUN npm run build

RUN serve build
```

The `npm install` command installs the required NPM packages, which includes packages that are only needed at build-time. The `npm run build` command uses the packages to create an optimized production-ready application build. Then, the container uses an HTTP server to expose the application by using the `serve` command.

If you build an image from the previous Containerfile, then the image contains both the build-time and runtime dependencies, which increases the image size. The resulting image also contains the Node.js runtime, which is not used at container runtime but might increase the attack surface of the container.

To avoid this issue, you can define two stages:

- **First stage:** Build the application.
- **Second stage:** Copy and serve the static files by using an HTTP server, such as NGINX or Apache Server.

The following example implements the two-stage build process.

```
# First stage
FROM registry.access.redhat.com/ubi8/nodejs-14:1 as builder (1)
COPY ./ /opt/app-root/src/
RUN npm install
RUN npm run build (2)

# Second stage
FROM registry.access.redhat.com/ubi8/nginx-120 (3)
COPY --from=builder /opt/app-root/src/ /usr/share/nginx/html (4)
```

1. Define the first stage with an alias. The second stage uses the `builder` alias to reference this stage.
2. Build the application.
3. Define the second stage without an alias. It uses the `ubi8/nginx-120` base image to serve the production-ready version of the application.
4. Copy the application files to a directory in the final image. The `--from` flag indicates that Podman copies the files from the `builder` stage.

Examine Container Data Layers

Container images use a copy-on-write (COW), layered file system. When you create a Containerfile, the `RUN`, `COPY`, and `ADD` instructions create *layers* (sometimes referred to as *blobs*).

The layered COW file system ensures that a container remains immutable. When you start a container, Podman creates and mounts a thin, ephemeral, writable layer on top of the container image layers. When you delete the container, Podman deletes the writable thin layer. This means that all container layers stay identical, except for the thin writable layer.

Cache Image Layers

Because all container layers are identical, multiple containers can share the layers. This means Podman can cache layers, and build only those layers that are modified or not cached.

You can use caching to decrease build time. For example, consider a Node.js application Containerfile:

```
...content omitted...
COPY . /app/
...content omitted...
```

The previous instruction copies every file and directory in the Containerfile directory to the `/app` directory in the container. This means that any changes to the application result in the rebuilding of every layer after the `COPY` layer.

You can change the instructions as follows:

```
...content omitted...
COPY package.json /app/
RUN npm ci --production
COPY src ./src
...content omitted...
```

This means that if you change your application source code in the `src` directory and rebuild your container image, then the dependency layer is cached and skipped, which reduces the build time. Podman rebuilds the dependency layer only when you change the `package.json` file.

Reduce Image Layers

You can reduce the number of container image layers, for example, by chaining instructions. Consider the following `RUN` instructions:

```
RUN mkdir /etc/gitea
RUN chown root:gitea /etc/gitea
RUN chmod 770 /etc/gitea
```

You can reduce the number of layers to one by chaining the commands. In Linux, you can chain commands by using the double ampersand (`&&`). You can also use the backslash character (`\`) to break a long command into multiple lines.

Consequently, the resulting `RUN` command looks as follows:

```
RUN mkdir /etc/gitea && \
    chown root:gitea /etc/gitea && \
    chmod 770 /etc/gitea
```

The advantage of chaining commands is that you create less container image layers, which typically results in smaller images.

However, chained commands are more difficult to debug and cache.

You can also create Containerfiles that do not use chained commands, and configure Podman to squash the layers. Use the `--squash` option to squash layers declared in the Containerfile. Alternatively, use the `--squash-all` option to also squash the layers from the parent image.

For example, consider the following three builds of one Containerfile.

```
[user@host ~]$ podman build -t localhost/not-squashed .
...output omitted...
[user@host ~]$ podman build --squash -t localhost/squashed .
...output omitted...
[user@host ~]$ podman build --squash-all -t localhost/squashed-all .
...output omitted...
[user@host ~]$ podman images --format="{{.Names}}\t{{.Size}}"
[localhost/not-squashed:latest]  419 MB (1)
[localhost/squashed:latest]      419 MB (2)
[localhost/squashed-all:latest]  394 MB (3)
```

1. The base image size is 419MB.
2. When Podman squashed the image layers, the image size stayed the same but the number of layers is lower.
3. When Podman squashed the layers of the container image and its parent image, the size reduced by 25MB.

Developers commonly reduce the number of layers by using multistage builds in combination with chaining some commands.

REFERENCES

podman-build(1) man page

Containerfile Documentation

Exploring the new Podman secret command

podman-secret-create(1) man page