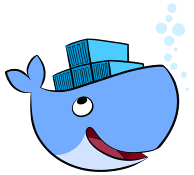
Docker for Python Developers

Fitter. Happier. More productive.



Presented by *Michael Herman* at dockercon 2018 SF

Note

These practices, from this presentation, can be used for any language. Examples are in Python, though - designed specifically for web developers and data scientists.

Agenda

Agenda

(1) Intro

1. About Me
2. Objectives
3. Why Docker?

Agenda

(1) Intro

1. About Me
2. Objectives
3. Why Docker?

(2) Best Practices

1. Keep images slim with Alpine
2. Use multi-stage builds
3. Order Dockerfile commands
4. Minimize the number of layers
5. Version Docker images
6. Create a non-root user
7. Do not store secrets in an image

Agenda

Agenda

Agenda

Intro

About Michael

```
$ whoami
michael.herman
```

About Michael

```
$ whoami
michael.herman
```

Day Job:

Software Engineer at ClickFox.

About Michael

```
$ whoami
michael.herman
```

Day Job:

Software Engineer at ClickFox.

Docker:

1. Avid Docker user since 2014.
2. Last year I architected and set up On-Demand Environments With Docker and AWS ECS.

About Michael

```
$ whoami
michael.herman
```

Day Job:

Software Engineer at ClickFox.

Docker:

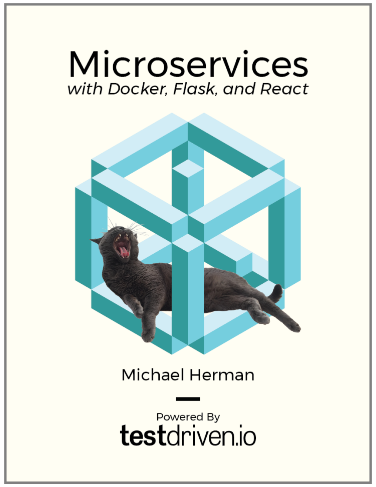
1. Avid Docker user since 2014.
2. Last year I architected and set up On-Demand Environments With Docker and AWS ECS.

Also:

1. Co-founder/author of Real Python
2. 😍 - tech writing/education, open source, financial models, radiohead

TestDriven.io

Started Testdriven.io late 2017...



Microservices with Docker, Flask, and React

Learn how to build, test, and deploy microservices powered by Docker, Flask, and React!

- Test-driven Development (TDD)
- AWS ECS, RDS, and Lambda
- React
- Blue/Green Deploys
- CI/CD

Objectives

By the end of this talk, you should be able to…

Objectives

By the end of this talk, you should be able to…

1. Use **Alpine Linux** based images as a starting point for your Dockerfiles

Objectives

By the end of this talk, you should be able to…

1. Use **Alpine Linux** based images as a starting point for your Dockerfiles
2. Create lean, production-ready images using **multi-stage builds**

Objectives

By the end of this talk, you should be able to…

1. Use **Alpine Linux** based images as a starting point for your Dockerfiles
2. Create lean, production-ready images using **multi-stage builds**
3. **Version** Docker images with Docker tags

Objectives

By the end of this talk, you should be able to…

1. Use **Alpine Linux** based images as a starting point for your Dockerfiles
2. Create lean, production-ready images using **multi-stage builds**
3. **Version** Docker images with Docker tags
4. **Order the statements** in your Dockerfile properly to avoid invalidating the cache

Objectives

By the end of this talk, you should be able to...

1. Use **Alpine Linux** based images as a starting point for your Dockerfiles
2. Create lean, production-ready images using **multi-stage builds**
3. **Version** Docker images with Docker tags
4. **Order the statements** in your Dockerfile properly to avoid invalidating the cache
5. **Reduce the number of layers** by logically combing `RUN` steps

Objectives

By the end of this talk, you should be able to...

1. Use **Alpine Linux** based images as a starting point for your Dockerfiles
2. Create lean, production-ready images using **multi-stage builds**
3. **Version** Docker images with Docker tags
4. **Order the statements** in your Dockerfile properly to avoid invalidating the cache
5. **Reduce the number of layers** by logically combing `RUN` steps
6. Run container processes as a **non-root user**

Objectives

By the end of this talk, you should be able to...

1. Use **Alpine Linux** based images as a starting point for your Dockerfiles
2. Create lean, production-ready images using **multi-stage builds**
3. **Version** Docker images with Docker tags
4. **Order the statements** in your Dockerfile properly to avoid invalidating the cache
5. **Reduce the number of layers** by logically combing `RUN` steps
6. Run container processes as a **non-root user**
7. Explain the best practices for **managing secrets** at both build and run-times

Objectives

By the end of this talk, you should be able to...

1. Use **Alpine Linux** based images as a starting point for your Dockerfiles
2. Create lean, production-ready images using **multi-stage builds**
3. **Version** Docker images with Docker tags
4. **Order the statements** in your Dockerfile properly to avoid invalidating the cache
5. **Reduce the number of layers** by logically combing `RUN` steps
6. Run container processes as a **non-root user**
7. Explain the best practices for **managing secrets** at both build and run-times
8. Use **Docker Compose** to build, run, and connect multiple containers together

Why Docker?

It simplifies…

Why Docker?

It simplifies…

- **Environment Setup**
    - Consistent environments and dependencies
    - Environment is up an running quickly with just a few key strokes

Why Docker?

It simplifies…

- **Environment Setup**
    - Consistent environments and dependencies
    - Environment is up an running quickly with just a few key strokes
- **Distributing and deploying code and applications**
    - You need access to not only the code, but the data and libraries used to create analysis from a notebook
    - Enables collaboration

Why Docker?

It simplifies…

- **Environment Setup**
    - Consistent environments and dependencies
    - Environment is up an running quickly with just a few key strokes
- **Distributing and deploying code and applications**
    - You need access to not only the code, but the data and libraries used to create analysis from a notebook
    - Enables collaboration

…so you can focus on the code and data analysis, etc.

Why Docker?

It simplifies...

- **Environment Setup**
  - Consistent environments and dependencies
  - Environment is up an running quickly with just a few key strokes
- **Distributing and deploying code and applications**
  - You need access to not only the code, but the data and libraries used to create analysis from a notebook
  - Enables collaboration

...so you can focus on the code and data analysis, etc.

Plus, it's much easier than dealing with virtual environments!

Best Practices

Keep images slim with Alpine (1)

Use Alpine Linux based images since they come only with the packages you need. Resulting images will be smaller.

Keep images slim with Alpine (1)

Use Alpine Linux based images since they come only with the packages you need. Resulting images will be smaller.

What are the benefits?

Keep images slim with Alpine (1)

Use Alpine Linux based images since they come only with the packages you need. Resulting images will be smaller.

What are the benefits?

1. Decreased hosting costs since less disk space is used
2. Quicker build, download, and run times
3. More secure (since there are fewer packages and libraries)
4. Faster deployments

Keep images slim with Alpine (1)

Use Alpine Linux based images since they come only with the packages you need. Resulting images will be smaller.

What are the benefits?

1. Decreased hosting costs since less disk space is used
2. Quicker build, download, and run times
3. More secure (since there are fewer packages and libraries)
4. Faster deployments

Web development example

```
FROM python:3.6-alpine

WORKDIR /app

COPY requirements.txt /
RUN pip install -r /requirements.txt  # flask and gunicorn

COPY . /app
```

Size before: 702MB, Size after: 102MB

Keep images slim with Alpine (2)

Keep images slim with Alpine (2)

Data science example

```
FROM python:3.6-alpine

RUN apk --no-cache add --virtual build-dependencies \
                build-base \
                python3-dev \
        && pip3 install \
                jupyter \
                pandas

WORKDIR /notebooks
```

Size before: 929MB, Size after: 634MB

Keep images slim with Alpine (2)

Data science example

```
FROM python:3.6-alpine

RUN apk --no-cache add --virtual build-dependencies \
                build-base \
                python3-dev \
        && pip3 install \
                jupyter \
                pandas

WORKDIR /notebooks
```

Size before: 929MB, Size after: 634MB

Want to inspect an image from Docker Hub?
Check out MicroBadger.

Use multi-stage builds (1)

Take advantage of multi-stage builds to create a temp image used for building an artifact that will be copied over to the production image. The temp build image is discarded along with the original files, folders, and dependencies associated with the image.

This produces a lean, production-ready image.

Use multi-stage builds (1)

Take advantage of multi-stage builds to create a temp image used for building an artifact that will be copied over to the production image. The temp build image is discarded along with the original files, folders, and dependencies associated with the image.

This produces a lean, production-ready image.

One use case is to use a non-Alpine base to install dependencies that require compilation. The wheel files can then be copied over to the final image.

Use multi-stage builds (1)

Take advantage of multi-stage builds to create a temp image used for building an artifact that will be copied over to the production image. The temp build image is discarded along with the original files, folders, and dependencies associated with the image.

This produces a lean, production-ready image.

One use case is to use a non-Alpine base to install dependencies that require compilation. The wheel files can then be copied over to the final image.

Web development example

```
FROM python:3.6 as base
COPY requirements.txt /
RUN pip wheel --no-cache-dir --no-deps --wheel-dir /wheels -r requirements.txt

FROM python:3.6-alpine
COPY --from=base /wheels /wheels
COPY --from=base requirements.txt .
RUN pip install --no-cache /wheels/* # flask, gunicorn, pycrypto
WORKDIR /app
COPY . /app
```

Size before: 705MB, Size after: 103MB

Use multi-stage builds (2)

Data science example

```
FROM python:3.6 as base
RUN pip wheel --no-cache-dir --no-deps --wheel-dir /wheels jupyter pandas

FROM python:3.6-slim
COPY --from=base /wheels /wheels
RUN pip install --no-cache /wheels/*
WORKDIR /notebooks
```

Size before: 929MB, Size after: 365MB

Use multi-stage builds (2)

Data science example

```
FROM python:3.6 as base
RUN pip wheel --no-cache-dir --no-deps --wheel-dir /wheels jupyter pandas

FROM python:3.6-slim
COPY --from=base /wheels /wheels
RUN pip install --no-cache /wheels/*
WORKDIR /notebooks
```

Size before: 929MB, Size after: 365MB

CI example

```
FROM python:3.6 as base
RUN pip wheel --no-cache-dir --no-deps --wheel-dir /wheels -r flask
COPY . /app
# What happens if the tests fail?
RUN py.test

FROM python:3.6-alpine
COPY --from=base /wheels /wheels
RUN pip install --no-cache /wheels/*
COPY . /app
```

Order Dockerfile commands

Docker caches the steps in a Dockerfile to speed up subsequent builds. When a change is made to a step, all steps following it will be redone.

Order Dockerfile commands

Docker caches the steps in a Dockerfile to speed up subsequent builds. When a change is made to a step, all steps following it will be redone.

Avoid invalidating the cache by-

1. Starting your Dockerfile with commands that are less likely to change
2. Putting commands that are more likely to change (like `COPY .`) as late as possible
3. Adding only the necessary files (use a *.dockerignore* file!)

Order Dockerfile commands

Docker caches the steps in a Dockerfile to speed up subsequent builds. When a change is made to a step, all steps following it will be redone.

Avoid invalidating the cache by-

1. Starting your Dockerfile with commands that are less likely to change
2. Putting commands that are more likely to change (like `COPY .`) as late as possible
3. Adding only the necessary files (use a *.dockerignore* file!)

Example

Order Dockerfile commands

Docker caches the steps in a Dockerfile to speed up subsequent builds. When a change is made to a step, all steps following it will be redone.

Avoid invalidating the cache by-

1. Starting your Dockerfile with commands that are less likely to change
2. Putting commands that are more likely to change (like `COPY .`) as late as possible
3. Adding only the necessary files (use a *.dockerignore* file!)

Example

```
FROM python:3.6-alpine

WORKDIR /app
# What happens when a change is made to sample.py?
COPY sample.py /app

COPY requirements.txt /
RUN pip install -r /requirements.txt  # flask and gunicorn
```

Order Dockerfile commands

Docker caches the steps in a Dockerfile to speed up subsequent builds. When a change is made to a step, all steps following it will be redone.

Avoid invalidating the cache by-

1. Starting your Dockerfile with commands that are less likely to change
2. Putting commands that are more likely to change (like `COPY .`) as late as possible
3. Adding only the necessary files (use a *.dockerignore* file!)

Example

```
FROM python:3.6-alpine

WORKDIR /app
# What happens when a change is made to sample.py?
COPY sample.py /app

COPY requirements.txt /
RUN pip install -r /requirements.txt  # flask and gunicorn
```

Move the `COPY sample.py /app` statement to the bottom!

Minimize the number of layers

Combine `RUN` steps that are related to prevent caching (since each `RUN` step will create a new layer) and using unnecessary disc space.

Minimize the number of layers

Combine `RUN` steps that are related to prevent caching (since each `RUN` step will create a new layer) and using unnecessary disc space.

Things to note

1. `RUN`, `COPY`, and `ADD` steps will create layers.
2. Each layer contains the differences from the previous layer.
3. Layers increase the size of the final image.

Minimize the number of layers

Combine `RUN` steps that are related to prevent caching (since each `RUN` step will create a new layer) and using unnecessary disc space.

Things to note

1. `RUN`, `COPY`, and `ADD` steps will create layers.
2. Each layer contains the differences from the previous layer.
3. Layers increase the size of the final image.

Tips

1. Put related commands (`apt-get update`/`install`) in the same `RUN` step.
2. Remove files in the same `RUN` step that created them.
3. Avoid using `apt-get upgrade` since it upgrades all packages to the latest version… Why?

Minimize the number of layers

Combine `RUN` steps that are related to prevent caching (since each `RUN` step will create a new layer) and using unnecessary disc space.

Things to note

1. `RUN`, `COPY`, and `ADD` steps will create layers.
2. Each layer contains the differences from the previous layer.
3. Layers increase the size of the final image.

Tips

1. Put related commands (`apt-get update`/`install`) in the same `RUN` step.
2. Remove files in the same `RUN` step that created them.
3. Avoid using `apt-get upgrade` since it upgrades all packages to the latest version… Why?

Finally…

Minimize the number of layers

Combine `RUN` steps that are related to prevent caching (since each `RUN` step will create a new layer) and using unnecessary disc space.

Things to note

1. `RUN`, `COPY`, and `ADD` steps will create layers.
2. Each layer contains the differences from the previous layer.
3. Layers increase the size of the final image.

Tips

1. Put related commands (`apt-get update`/`install`) in the same `RUN` step.
2. Remove files in the same `RUN` step that created them.
3. Avoid using `apt-get upgrade` since it upgrades all packages to the latest version... Why?

Finally...

Use multi-stage builds as much as possible!

Version Docker images

If you rely on just using the `latest` tag, you'll have no real way of knowing which image version is actually running in a specific environment.

Version Docker images

If you rely on just using the `latest` tag, you'll have no real way of knowing which image version is actually running in a specific environment.

How?

Version Docker images

If you rely on just using the `latest` tag, you'll have no real way of knowing which image version is actually running in a specific environment.

How?

1. timestamps
2. image ids
3. Git commit hashes

Version Docker images

If you rely on just using the `latest` tag, you'll have no real way of knowing which image version is actually running in a specific environment.

How?

1. timestamps
2. image ids
3. Git commit hashes

Example

You could use both the git commit SHA1 hash (to associate the image back to a specific commit to help with debugging) along with and the environment name:

`/$PROJECT/$ENVIRONMENT:$SHA1`

```
$ docker build -t web/prod:a072c4e5d94b5a769225f621f08af3d4bf820a07 .
```

Create a non-root user

By default, Docker runs container processes as root inside of a container. This is a bad practice since attackers can gain root access to the Docker host if they manage to break out of the container.

If you're root in the container, you'll be root on the host.

Create a non-root user

By default, Docker runs container processes as root inside of a container. This is a bad practice since attackers can gain root access to the Docker host if they manage to break out of the container.

If you're root in the container, you'll be root on the host.

Examples

1. Web Development
2. Data Science

Create a non-root user

By default, Docker runs container processes as root inside of a container. This is a bad practice since attackers can gain root access to the Docker host if they manage to break out of the container.

If you're root in the container, you'll be root on the host.

Examples

1. Web Development
2. Data Science

Verify

```
$ docker run -p 5000:5000 -i sample id
uid=100(app) gid=101(app) groups=101(app)

$ docker run -p 8888:8888 -i ds id
uid=100(app) gid=101(app) groups=101(app)
```

(The Linux `id` command displays info about the current user.)

Now, each containers' user and associated group are non-root users.

Do not store secrets in an image (1)

Do not store secrets in an image (1)

Secrets?

Do not store secrets in an image (1)

Secrets?

Sensitive info. Database credentials, SSH keys, usernames and passwords, API tokens, etc.

Do not store secrets in an image (1)

Secrets?

Sensitive info. Database credentials, SSH keys, usernames and passwords, API tokens, etc.

Secret Management (build vs run time)

Do not store secrets in an image (1)

Secrets?

Sensitive info. Database credentials, SSH keys, usernames and passwords, API tokens, etc.

Secret Management (build vs run time)

At run-time (recommended!):

1. You can pass secrets in via env variables, but they will be visible in all child processes - i.e., linked containers, `docker inspect` - and logs. It's also difficult to update them.

   ```
   $ docker run -p 5000:5000 -d -e "foo=bar" -i sample \
   gunicorn -b 0.0.0.0:5000 sample:app

   f8650976bcb9a50257aa9c39114207bb07d42d89f9ae00f5f2ba36c68fc

   $ docker inspect f8650976bcb9a50257aa9c39114207bb07d42d89f9ae00f5f2ba36c68fc
   ```

2. Passing them in using a shared volume is a better solution, but they should be encrypted (via Vault or KMS) since they are saved to disc.

Do not store secrets in an image (2)

At Build time:

Do not store secrets in an image (2)

At Build time:

1. Never store secrets in a Docker image that will be pushed to a public repo.
2. You can pass secrets in at build-time using build-time args, but they will be visible to those who have access to the image via `docker history`.

   Dockerfile:

   ```
   FROM alpine
   ARG foo
   RUN echo "Hello, World!"
   ```

   Example:

   ```
   $ docker build --build-arg "foo=bar" -t hi .
   Successfully built f2bcff49ac09

   $ docker history f2bcff49ac09 | grep foo
   1796fafd3d00        About a minute ago      /bin/sh -c #(nop)  ARG foo
   ```

Do not store secrets in an image (2)

At Build time:

1. Never store secrets in a Docker image that will be pushed to a public repo.
2. You can pass secrets in at build-time using build-time args, but they will be visible to those who have access to the image via `docker history`.

   Dockerfile:

   ```
   FROM alpine
   ARG foo
   RUN echo "Hello, World!"
   ```

   Example:

   ```
   $ docker build --build-arg "foo=bar" -t hi .
   Successfully built f2bcff49ac09

   $ docker history f2bcff49ac09 | grep foo
   1796fafd3d00        About a minute ago       /bin/sh -c #(nop)  ARG foo
   ```

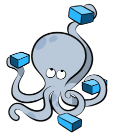
!!! Using Docker Swarm? Check out Docker Secrets. !!!

Docker Compose

Docker Compose

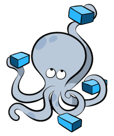
Docker Compose

Docker Compose is an orchestration tool used for running multi-container apps. It helps streamline building, running, and connecting multiple containers together. To use Docker Compose, you'll need to define how to build your containers with YAML in a *docker-compose.yml* file.



Docker Compose

Docker Compose is an orchestration tool used for running multi-container apps. It helps streamline building, running, and connecting multiple containers together. To use Docker Compose, you'll need to define how to build your containers with YAML in a *docker-compose.yml* file.



Examples

1. Web Development
2. Data Science

Use Case

Use Case

Use Case

Scenario

Use Case

Scenario

1. Ship dependencies along with the code to secure environments cut off from the internet
2. Multiple operating systems
3. Python 2.6, 2.7, 3.6

Use Case

Scenario

1. Ship dependencies along with the code to secure environments cut off from the internet
2. Multiple operating systems
3. Python 2.6, 2.7, 3.6

Docker

Use Case

Scenario

1. Ship dependencies along with the code to secure environments cut off from the internet
2. Multiple operating systems
3. Python 2.6, 2.7, 3.6

Docker

1. Spin up containers that match the requirements
2. Build the wheel files
3. Bundle code + wheels

Use Case

Scenario

1. Ship dependencies along with the code to secure environments cut off from the internet
2. Multiple operating systems
3. Python 2.6, 2.7, 3.6

Docker

1. Spin up containers that match the requirements
2. Build the wheel files
3. Bundle code + wheels

https://realpython.com/offline-python-deployments-with-docker

That's it!

What's next?

That's it!

What's next?

Resources

1. Slides - http://mherman.org/presentations/dockercon-2018
2. Repo - https://github.com/testdrivenio/docker-python-devs
3. 7 best practices for building containers
4. How to Build 12 Factor Microservices on Docker
5. Docker Cheat Sheet
6. Simplifying Offline Python Deployments With Docker
7. From Docker:
    - Best practices for writing Dockerfiles
    - Docker development best practices

That's it!

What's next?

Resources

1. Slides - http://mherman.org/presentations/dockercon-2018
2. Repo - https://github.com/testdrivenio/docker-python-devs
3. 7 best practices for building containers
4. How to Build 12 Factor Microservices on Docker
5. Docker Cheat Sheet
6. Simplifying Offline Python Deployments With Docker
7. From Docker:
    - Best practices for writing Dockerfiles
    - Docker development best practices

Questions?

✌️