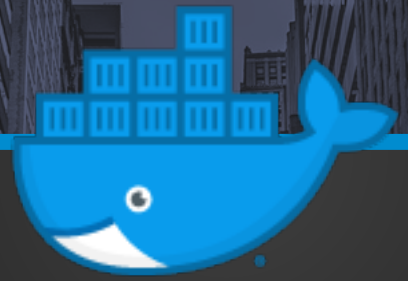
kubernetes fundamentals

docker deep dive

Advanced containers



Advanced containers – interactive shell

- With a running container, you can attach to be able to interact with it.
- Docker attach <container-name>
- You can now get it's output (like with docker logs), provide input if necessary
- To detach without stopping it, use the detach sequence (hold CTRL, press p then q)

Advanced containers – exec

- With a running container, you can execute any arbitrary process available on the container.
- Frequently you can use this to “log in” to a shell. This is dependent on the image the container was built from.
- This is much like SSH connection to a server
- Once inside, you have free reign to look around and inspect your container
- This is very useful for debugging. You’ve built an image and you’re not sure if the image looks like you expect. So you can get into a running container and view things. For example, when you built your nginx image, how did you know the file was there until you tried the web site?
- The location of the shell process depends on the container.
- `docker exec -it nginx bash`

Advanced containers – exec lab

- Start a new container from the nginx image on Docker Hub
- Using exec, get into the container and have a look around
- Visit the web site and make sure you see the default home page.
- Now go change the home page index.html file located in /usr/share/nginx/html.
  - Uh oh, how do you edit the file?
  - Try installing vim. This is a Debian OS, so use `apt-get install vim`. First update the package manager with `apt-get update`
  - Now update the headline
- Verify that your changes show in the web site.

- `docker exec -it <container name> bash`

Advanced containers – export

- The docker export command dumps a container's filesystem to a tar archive
- Useful for debugging or getting quick access to the files in a container
- `docker export nginx > nginx.tar`

Advanced containers – diff

- The docker diff command will show you filesystem changes from the base image
- Useful to see what a container does after it's been started
- Docker diff <container name>
- A = added
- C = changed
- D = deleted

Advanced containers – cp

- Allows you to copy files to/from a Docker container to/from the host computer.
- Format is:
  - `docker cp <container name>:<path> <host path>`
  - `docker cp <host path> <container name>:<path>`

Advanced containers – inspect

- You can inspect the details on a container. Everything you'd need to know about it is here.
- Inspect also works for other Docker resources, like images and networks.

Advanced containers – kill

- Unlike docker stop, which will attempt to stop a container process gracefully, docker kill will send a SIGKILL signal to the container's running process.
- This is a bit more brute force.
- Typically this isn't needed, as a docker stop will send a SIGKILL automatically if it doesn't shut down gracefully.

Advanced containers – pause/unpause

- Unlike stop or kill, this will pause the processes inside a running container.
- Perhaps it's vital to stop the processing on a container, but you need to investigate further. Pausing allows that. You can put things on hold, get your bearings, and then unpause or stop the container as needed.
- `docker ps` shows the status of a container as paused, but it's not completely obvious.
- You unfortunately cannot exec into a bash shell in a paused container for investigation.
- `docker pause <container>`
- You can resume with `docker unpause <container>`

Advanced containers – rename

- Rename a container with `docker rename <old> <new>`

Advanced containers – restarts

- To restart a running container, just use `docker restart <container>`
- When you run a container, you can specify a restart policy with `--restart`
  - No – the default. Do not restart a stopped container
  - On-failure – restart if it failed due to an error
  - Unless-stopped – restart unless it is explicitly stopped or Docker itself is stopped or restarted
  - Always – always restart it, even if Docker itself is restarted. Good for containers that must survive a server reboot.

Advanced containers – stats

- Useful to see what Docker processes are running on the HOST and what their resource usage is.
- Works much like the top Linux command.
- `docker stats`

Advanced containers – top

- To see the the running processes for a container, you can do this directly without logging in to an interactive shell on a container.
- Gives you an output similar to ps inside the container.
- Despite the name, it doesn't give resource information and isn't live updating like top is.
- `docker top <container>`

Advanced containers – update

- Sometimes you want to update the configuration for a container after it has been started with a docker run command.
- `docker update` is the command.
- There are many options.
- For example, if you want to update the restart policy of a container:
  - `docker update --restart always nginx`

Advanced containers – a word on ports

- You've learned how to specify port mappings when creating a container with `docker run -p`
- Unfortunately, there isn't a way to update port mappings after creation. Why? The world may never know.
- The workaround is to stop and remove the container and use a new run command with the proper ports.
- There are also some options for messing with networks, but that beyond the scope of this course.
- Note that since we're going to get into orchestration soon, this isn't really a big problem.

Advanced containers – prune

- Sometimes you'll build up a lot of stopped containers that just clutter up your system.
- `Docker container prune` allows you to clean them up in one command.
- Just be careful, as this will in one swipe clear them all! If you have any that you need, you'll lose them!

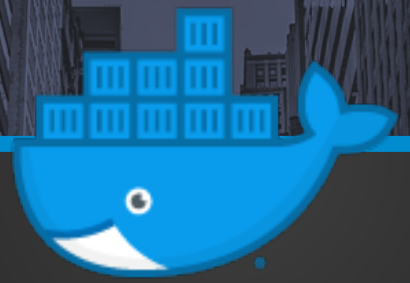
Advanced containers – linking

- This is a legacy feature, but still useful to know. Networks (next) are the preferred way of communicating.
- You can explicitly link containers to each other so that they can communicate by name.
- You link containers by including a --link option in the run command and provide the container name

Advanced containers – networks

- You can create Docker networks to allow containers to easily communicate.
- Types:
  - Bridge (default) – this one is created by Docker and containers go here by default. Not recommended for production. No container communication by name.
  - Bridge (user-defined) – Recommended for standalone containers running on the same host. Containers can communicate by name.
  - Overlay – allows networking across Docker hosts. Works with Swarm.
- Networks are managed with the `docker network …` commands
- Containers are put into a network with the `--network` option in the `docker run` command.
- Existing containers can be put into a network with `docker network connect <network name> <container name>`

Networks: lab

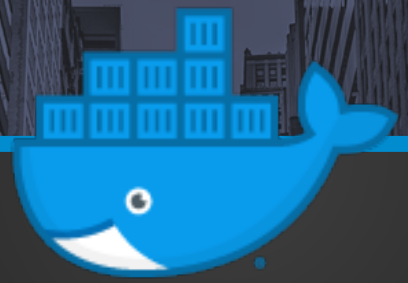


Networks – lab

Let's test out creating a network, putting containers into it, and have them communicate by name.

1. Create a Docker bridge network called alpine-net
2. Start up three Alpine containers. `docker run -dit --name=name alpine ash`
    1. For alpine1, include the network in the run command.
    2. For alpine2, include the network in the run command.
    3. For alpine3, do not include the network.
3. Inspect the bridge and alpine-net networks. View the containers and IP addresses.
4. Attach to alpine1 and attempt to ping alpine2 and alpine3 by both name and IP. What results did you get?
5. Detach (don't exit) from alpine1, and now connect alpine3 to your network.
6. Repeat step 4. How does it differ?
7. When done, remove the containers and network.

Advanced images



Advanced images – tagging

- You can tag images after creation, or to add new tags to an existing image
- You can tag based on an existing tag or image ID
- Could be useful for tagging an image long after building for upload to a private repository by including the URL in front.
  - For example, `docker tag 0e5574283393 companyregistry:5000/comp/myapp:v1`

Advanced images – images from archive

- Images can be saved to physical files or loaded from files
- Useful for archiving or transporting in a way different from the normal Docker registry mechanism.
- Images are saved to tar files.
- Name and tag are preserved in the tar file.
- Docker save <image name> > file.tar
  - Exports the image off to a file
- Docker load < file.tar
  - Loads the image back into your list of images

Advanced images – images archive lab

- Save an image from your machine to a tar file. Try using busybox:latest from Docker Hub.
- Now remove the image from your machine (not the file)
- Load the image back from the file
- Verify that the image is present again

Advanced images – history

- Shows you the history of changes to an image
- `docker history <image name>`

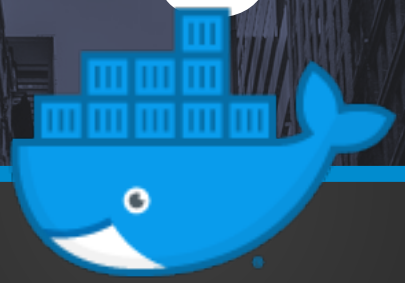
Advanced images – push/pull

- Pull allows you to download an image from a registry without running it. Good for either public (Docker Hub) or private registries.
- Push to upload your image to a registry.
- To connect to private registries, you must use `docker login` to authenticate.

Advanced images – prune

- Like Docker container prune, docker image prune allows you to clean up unused images in one command. Useful if your machine is loaded down with unused contaners.
- There are command options that allow you to force the removal of all images, not just those unused by containers.

Private registry: lab



Private registry – lab

Let's simulate having a private registry by starting one on our local workstation and then work with it. It runs as a Docker container!

1. Find the registry image on Docker Hub. Review the instructions and run it on your machine.
2. Now try tagging an image and pushing it to your new private registry. Perhaps you can pull busybox from Docker Hub and retag it? Try it.

Note that there's more involved with setting up a production grade registry, such as authentication. If you're working with a secured registry, you just need to login with `docker login`.

Advanced image construction

Advanced image construction - overview

- We've learned that images are made of layers. These layers so far have been simple. But there are more advanced ways to build images.
- Organizations can have standardized images that are intended to be inherited by teams. There can be multiple layers here.
  - For example, Company A might standardize on Ubuntu 18.10. The enterprise architecture team will choose this specific image and tag from Docker Hub. They may make adjustments and create CompanyA/ubuntu.
  - Company A's security team wants a list of security patches and settings applied to all images in the company. They create a standard image FROM CompanyA/ubuntu. They create CompanyA/ubuntu-secure.
  - The architecture team wants to standardize on Java 8 for all Java projects. They create CompanyA/java8 FROM CompanyA/ubuntu-secure.
  - The ecommerce team creates their project. They'll create a variety of images FROM CompanyA/java8.
  - In this way, every time any of these images are updated, subsequent images will pick up the latest changes. For example, the security team updates their image with a new security fix. Then when the ecommerce team does another build/deploy, they'll pick up that security fix.

Advanced image – create from container

- We built images from Dockerfiles. However, this is not the only way.
- You can create a new container, make changes, install software, etc, and then create an image based on that container.
- While possible, this isn't the recommended approach and you should approach it with caution. This means you've moved away from infrastructure-as-code!
- However, this can be useful for small experiments, proofs-of-concepts, and personal utilities, such as a client for operating a CLI on a remote service like AWS.
- The procedure roughly looks like:
  - docker create … (this creates a new container)
  - docker start … (this starts the container you just created)
  - Make changes, install software, etc.
  - docker commit … (this creates a new image without a tag)
  - docker tag … (you'll now tag this image using it's ID)

Image from container – lab

Let's create an image from a container. We'll start with a base image from Docker Hub. Then customize it, and finally create an image from that.

1. First, create a new container using the Nginx image
2. Run it, then login to it and make some changes to the home page. /usr/share/nginx/html/index.html
3. Then bake an image from that container.
4. Now delete the container.
5. Finally, create a running container from your new image and verify that it serves the home page you updated.

Advanced image – multi-stage

- This is a recent addition to Docker
- Before this, you had to employ custom tricks and hacks to build different versions of an application image.
  - For example, you might want an image with extra software for dev environment, but this removed for production.
- With multi-stage, you can have multiple FROM statements in one Dockerfile. You can copy artifacts from one stage to another. This simplifies the process and reduces errors.

```
FROM golang:1.7.3 as builder
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /go/src/github.com/alexellis/href-counter/app .
CMD ["./app"]
```

Advanced image – alpine

- A goal of image creation should be reduction of image size.
- Some images can be very large. Many images in Docker Hub exceed 1gb.
- Additionally, many standard distros contain security vulnerabilities. If you remove all unnecessary components, you reduce security flaws.
- Many Docker users choose to use Alpine Linux, which is an extremely stripped down Linux distribution. You get the bare bones Linux.
- Alpine is only 4-5 mb in size! Additionally, the runtime footprint is less.
- Due to the small size, this means deploying to a server is very fast with the small download size.
- Alpine is built to be secure. This does not however mean you should just assume it's good enough. Security teams should still
- You'll need to install any necessary software in your Dockerfile. You don't even get curl.

Advanced image – other minimal images

- Newer images now exist for small images that bring the same benefits of Alpine, but in other possibly more friendly Linux distros. They offer a bit more balance of size and functionality than Alpine.
  - Ubuntu Minimal 18.04 is 29 mb
  - Debian Slim is around 88 mb

Advanced image – alpine lab

- Time to build an image based on Alpine. Because this is stripped down Linux, you'll need to install your own software components.
- Create a Dockerfile with a FROM alpine line
- Install some software. Perhaps curl. You'll need to run as USER root or use sudo in your commands.
- The package manager is apk
  - sudo apk install nginx

Advanced image – running as root

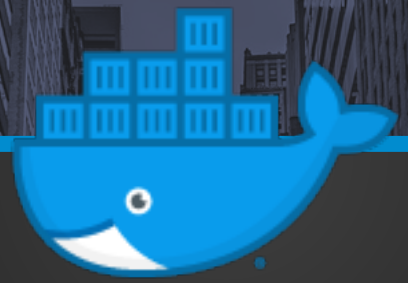
- Well designed systems adhere to the principle of least privilege. This applies to containers as well.
- By default, Docker containers run with the root user inside the container
- This opens a host of security vulnerabilities. In fact, you can even open up the host system to security breaches!
- Since most container workloads don't need root access, don't give it to them.
- Let's update our Dockerfile so that we create a new user and then use it going forward.

```
FROM <base image>

RUN groupadd -g 999 appuser && \
      useradd -r -u 999 -g appuser appuser
USER appuser

... <rest of Dockerfile> ...
```

Advanced image – running as root

- What if you're using someone else's image that didn't specify a non-root user?
  - You can create your image image using that image in the FROM line and do your own user management as before.
  - You can optionally run a container with the user specified at runtime: `docker run --user 1001 <rest of run command>`
- But when creating your own images, it's best to follow best practices and run your containers as non-root users in the Dockerfile. This way non-root user is the default.

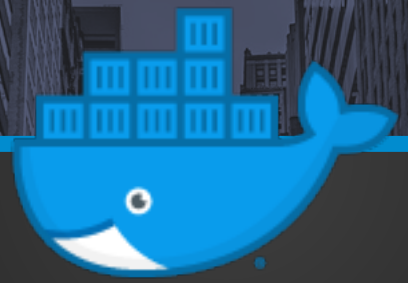
Ci/cd pipeline



Ci/cd pipeline – overview

- A CI/CD pipeline looks like this:
  - Build your application (pre-Docker)
  - Build your Docker image
  - Push your Docker image to a repository like AWS Elastic Container Registry (ECR)
  - Update your running application to use your new image.
- Let's take a look at a real example:
  - `/Users/tsolley/Documents/VergeOps/rabble/feature-service/jenkinsfile`
  - Let's also take a look at the Jenkins server
  - This example uses AWS ECS instead of Kubernetes as the orchestrator.
  - Note that there is a separate Jenkins build that does automated testing. It is dependent on this build's completion. Once this build completes, there's a quiet period (to allow deployment to complete), then the automated tests will run and issue reports which are stored in Jenkins as Cucumber reports.

Docker compose



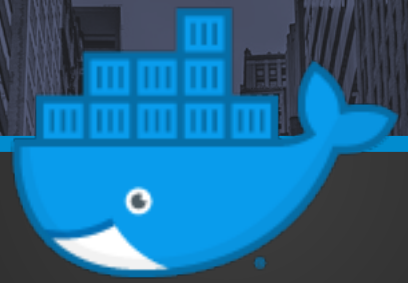
Docker compose – overview

- We've seen how we can run individual Docker containers. But there can and often are lots of options to specify. And running a whole bunch of containers to run our application would be a real drag. Scripting it out seems hacky. Is there a better way?
- What if you want a lightweight way to run your application without the overhead of a Kubernetes application?
- Compose is a tool for running multiple related containers together with one command. It is included with Docker by default.
- You specify all the configurations in a YAML file. Another example of infrastructure-as-code!
- Compose creates a new network for your containers so that they can communicate with each other by name. Or you can specify your own.
- You can refer to an existing image in Docker Hub, on your local machine, or a private repo, or you can refer to a Dockerfile location, and Compose will build the image for you before running the container.
- To start up your app, just use `docker-compose up`. To stop it, use `docker-compose down`.

Docker compose – overview

```
version: '3'
services:
  app:
    build: path/to/Dockerfile
    restart: always
    ports:
    - "80:80"
  db:
    image: ”db:1.0"
    restart: always
    ports:
    - ”3306:3306"
    environment:
    - MY_ENV=my_value
```

Docker compose:



lab

Docker compose – wordpress lab

- Build a new Docker Compose file
- Use two Docker Hub images. One for Wordpress, and the other a Mysql image. You'll need to review the documentation for the images to find out what important options to pass the containers.
- Bring the stack up and try to use the blog.

wrap up