**Container Scenarios**

**Processes:**

* Containers are just normal Linux Processes with additional configuration applied.

**Process Directory:**

* The configuration for each process is defined within the /proc directory. If you know the process ID, then you can identify the configuration directory.
* Command:
  + DBPID=$(pgrep redis-server)
  + echo Redis is $DBPID
  + ls /proc
  + ls /proc/$DBPID
  + cat /proc/$DBPID/environ
  + docker exec -it db env

**Namespaces:**

* One of the fundamental parts of a container is namespaces. The concept of namespaces is to limit what processes can see and access certain parts of the system, such as other network interfaces or processes.
* When a container is started, the container runtime, such as Docker, will create new namespaces to sandbox the process. By running a process in it's own Pid namespace, it will look like it's the only process on the system.
* The available namespaces are:
  + Mount (mnt)
  + Process ID (pid)
  + Network (net)
* Interprocess Communication (ipc)
  + UTS (hostnames)
  + User ID (user)
  + Control group (cgroup)
* More information:
  + <https://en.wikipedia.org/wiki/Linux_namespaces>

## **Unshare can launch "contained" processes.**

* Without using a runtime such as Docker, a process can still operate within it's own namespace. One tool to help is unshare.
* unshare --help
* With unshare it's possible to launch a process and have it create a new namespace, such as Pid. By unsharing the Pid namespace from the host, it looks like the bash prompt is the only process running on the machine.
* sudo unshare --fork --pid --mount-proc bash
* ps
* exit

**What happens when we share a namespace?**

* Under the covers, Namespaces are inode locations on disk. This allows for processes to share/reused the same namespace, allowing them to view and interact.
* List all the namespaces
  + ls -lha /proc/$DBPID/ns/
* Another tool, NSEnter is used to attach processes to existing Namespaces. Useful for debugging purposes.
  + nsenter --help
  + nsenter --target $DBPID --mount --uts --ipc --net --pid ps aux
* With Docker, these namespaces can be shared using the syntax container:<container-name>

**Chroot:**

* An important part of a container process is the ability to have different files that are independent of the host. This is how we can have different Docker Images based on different operating systems running on our system.
* Chroot provides the ability for a process to start with a different root directory to the parent OS. This allows different files to appear in the root

**Cgroup (Control Groups)**

* CGroups limit the amount of resources a process can consume. These cgroups are values defined in particular files within the /proc directory.
* To see the mappings, run the command:
  + cat /proc/$DBPID/cgroup
* These are mapped to other cgroup directories on disk a
  + ls /sys/fs/cgroup/

**What are the CPU stats for a process**

* The CPU stats and usage is stored within a file too!
  + cat /sys/fs/cgroup/cpu,cpuacct/docker/$DBID/cpuacct.stat
* The CPU shares limit is also defined here.
  + cat /sys/fs/cgroup/cpu,cpuacct/docker/$DBID/cpu.shares
* All the Docker cgroups for the container's memory configuration are stored within:
  + ls /sys/fs/cgroup/memory/docker/
* Each of the directory is grouped based on the container ID assigned by Docker.
  + DBID=$(docker ps --no-trunc | grep 'db' | awk '{print $1}')
  + WEBID=$(docker ps --no-trunc | grep 'nginx' | awk '{print $1}')
  + ls /sys/fs/cgroup/memory/docker/$DBID

**How to configure cgroups?**

* One of the properties of Docker is the ability to control memory limits. This is done via a cgroup setting.
* By default, containers have no limit on the memory. We can view this via the docker stats command.
  + docker stats db --no-stream
* The memory quotes are stored in a file called memory.limit\_in\_bytes.
  + echo 8000000 > /sys/fs/cgroup/memory/docker/$DBID/memory.limit\_in\_bytes
* If you read the file back, you'll notice it's been converted to 7999488.
  + cat /sys/fs/cgroup/memory/docker/$DBID/memory.limit\_in\_bytes
* When checking Docker Stats again, the memory limit of the process is now 7.629M
  + docker stats db --no-stream

**Seccomp / AppArmor**

* All actions with Linux are done via syscalls. The kernel has 330 system calls that perform operations such as read files, close handles and check access rights. All applications use a combination of these system calls to perform the required operations.
* AppArmor is an application defined profile that describes which parts of the system a process can access.
* To view the current AppArmor profile assigned to a process via.
  + cat /proc/$DBPID/attr/current
* The default AppArmor profile for Docker is docker-default (enforce).
* Prior to Docker 1.13, it stored the AppArmor Profile in /etc/apparmor.d/docker-default (which was overwritten when Docker started, so users couldn't modify it. After v1.13, Docker now generates docker-default in tmpfs, uses apparmor\_parser to load it into the kernel, then deletes the file.
  + Template
    - <https://github.com/moby/moby/blob/a575b0b1384b2ba89b79cbd7e770fbeb616758b3/profiles/apparmor/template.go>
* Seccomp provides the ability to limit which system calls can be made, blocking aspects such as installing Kernel Modules or changing the file permissions.
* The default allowed calls with Docker can be found at <https://github.com/moby/moby/blob/a575b0b1384b2ba89b79cbd7e770fbeb616758b3/profiles/seccomp/default.json>
* When assigned to a process it means the process will be limited to a subset of the ability system calls. If it attempts to call a blocked system call it will receive the error "Operation Not Allowed".
* The status of SecComp is also defined within a file.
  + cat /proc/$DBPID/status
  + cat /proc/$DBPID/status | grep Seccomp

**Capabilities**

* Capabilities are groupings about what a process or user has permission to do. These Capabilities might cover multiple system calls or actions, such as changing the system time or hostname.
* The status file also contains the Capabilities flag. A process can drop as many Capabilities as possible to ensure it's secure
  + cat /proc/$DBPID/status | grep ^Cap
* The flags are stored as a bitmask that can be decoded with capsh
  + capsh --decode=00000000a80425fb

**Container Image**

* A container image is a tar file containing tar files. Each tar file is a layer. Once all tar files have been extract into the same location then you have the container's filesystem
* Pull the layers onto your local system
  + docker pull redis:3.2.11-alpine
* Export the image into the raw tar format.
  + docker save redis:3.2.11-alpine > redis.tar
* Extract to the disk
  + tar -xvf redis.tar
* All of the layer tar files are now viewable.
  + ls
* The image also includes metadata about the image, such as version information and tag names.
  + cat repositories
  + cat manifest.json
* Extracting a layer will show you which files that layer provides.
  + tar -xvf da2a73e79c2ccb87834d7ce3e43d274a750177fe6527ea3f8492d08d3bb0123c/layer.tar
* Creating empty image
  + tar cv --files-from /dev/null | docker import - empty
* By importing the tar, the additional metadata will be created.
  + docker images
* Creating Image without Dockerfile
  + The previous idea of importing a Tar file can be extended to create an entire image from scratch.
  + Docker provide a script to download the BusyBox rootfs at <https://github.com/moby/moby/blob/a575b0b1384b2ba89b79cbd7e770fbeb616758b3/contrib/mkimage/busybox-static>
  + curl -LO https://raw.githubusercontent.com/moby/moby/a575b0b1384b2ba89b79cbd7e770fbeb616758b3/contrib/mkimage/busybox-static && chmod +x busybox-static
  + ./busybox-static busybox
  + Running the script will download the rootfs and main binaries.
  + ls -lha busybox
  + The default Busybox rootfs doesn't include any version information so let's create a file.
    - echo KatacodaPrivateBuild > busybox/release
  + As before, the directory can be converted into a tar and automatically imported into Docker as an image.
    - tar -C busybox -c . | docker import - busybox
  + This can now be launched as a container.
    - docker run busybox cat /release

**Docker:**

* Docker describes themselves as "an open platform for developers and sysadmins to build, ship, and run distributed applications".
* Docker allows you to run containers. A container is a sandboxed process running an application and its dependencies on the host operating system. The application inside the container considers itself to be the only process running on the machine while the machine can run multiple containers independently.

**Docker Example:**

* Scenario:Jane, a developer who needs to deploy a new Key-Value Store for an application she's working with. After discussions, it's been decided to use Redis, a popular KV Store.
* Step1: Running a Container
  + To find an image for Redis.
    - docker search redis
  + By default, Docker will run a command in the foreground. To run in the background, the option *-d* needs to be specified
    - Docker run -d redis
  + By default, Docker will run the *latest* version available. If a particular version was required, it could be specified as a tag, for example, version 3.2 would be *docker run -d redis:3.2*.
* Step2: finding Running Containers
  + To list all the running containers.
    - docker ps
    - docker inspect <friendly-name|container-id>
    - docker logs <friendly-name|container-id>
* Step3: Accessing Redis
  + Jane is happy that *Redis* is running, but is surprised that she cannot access it. The reason is that each container is sandboxed. If a service needs to be accessible by a process not running in a container, then the port needs to be exposed via the Host
  + Once exposed, it is possible to access the process as if it were running on the host OS itself.
  + default other applications and libraries expect a *Redis* instance to be listening on the port.
  + After reading the documentation, Jane discovers that ports are bound when containers are started using the *-p <host-port>:<container-port>* option. Jane also discovers that it's useful to define a name when starting the container, this means she doesn't have to use Bash piping or keep looking up the name when trying to access the logs.
  + Jane finds the best way to solve her problem of running *Redis* in the background, with a name of *redisHostPort* on port *6379* is using the following command
    - docker run -d --name redisHostPort -p 6379:6379 redis:latest
  + By default, the port on the host is mapped to 0.0.0.0, which means all IP addresses. You can specify a particular IP address when you define the port mapping, for example, *-p 127.0.0.1:6379:6379*
* Step4: Accessing Radis
  + The problem with running processes on a fixed port is that you can only run one instance. Jane would prefer to run multiple *Redis* instances and configure the application depending on which port Redis is running on.
  + After experimenting, Jane discovers that just using the option *-p 6379* enables her to expose *Redis* but on a randomly available port. She decides to test her theory using
    - docker run -d --name redisDynamic -p 6379 redis:latest
  + While this works, she now doesn't know which port has been assigned. Thankfully, this is discovered via
    - docker port redisDynamic 6379
  + Jane also finds that listing the containers displays the port mapping information,
    - docker ps
* Step5: Persisting Data
  + After working with containers for a few days, Jane realises that the data stored keeps being removed when she deletes and re-creates a container. Jane needs the data to be persisted and reused when she recreates a container.
  + Containers are designed to be stateless. Binding directories (also known as volumes) is done using the option *-v <host-dir>:<container-dir>*. When a directory is mounted, the files which exist in that directory on the host can be accessed by the container and any data changed/written to the directory inside the container will be stored on the host. This allows you to upgrade or change containers without losing your data.
  + Using the Docker Hub documentation for [Redis](https://hub.docker.com/_/redis/), Jane has investigated that the official Redis image stores logs and data into a /data directory.
  + Any data which needs to be saved on the Docker Host, and not inside containers, should be stored in */opt/docker/data/redis*.
  + Command
    - docker run -d --name redisMapped -v /opt/docker/data/redis:/data redis
  + Docker allows you to use $PWD as a placeholder for the current directory.
* Step6: Running a container in the foreground.
  + Jane has been working with Redis as a background process. Jane wonders how containers work with foreground processes, such as *ps* or *bash*.
  + Previously, Jane used the *-d* to execute the container in a detached, background, state. Without specifying this, the container would run in the foreground. If Jane wanted to interact with the container (for example, to access a bash shell) she could include the options *-it*.
  + As well as defining whether the container runs in the background or foreground, certain images allow you to override the command used to launch the image. Being able to replace the default command makes it possible to have a single image that can be re-purposed in multiple ways. For example, the Ubuntu image can either run OS commands or run an interactive bash prompt using */bin/bash*
  + Command launches an Ubuntu container and executes the command *ps* to view all the processes running in a container.
    - docker run ubuntu ps
  + To access a bash shell inside of a container.
    - docker run -it ubuntu bash

**Deploy Static HTML Website as Container:**

* Step1: Create DockerFile
  + Docker Images start from a base image. The base image should include the platform dependencies required by your application, for example, having the JVM or CLR installed.
  + This base image is defined as an instruction in the Dockerfile. Docker Images are built based on the contents of a Dockerfile. The Dockerfile is a list of instructions describing how to deploy your application.
  + In this example, our base image is the Alpine version of Nginx. This provides the configured web server on the Linux Alpine distribution.
  + Docker Files:
    - FROM nginx:alpine
    - COPY . /usr/share/nginx/html
    - The first line defines our base image. The second line copies the content of the current directory into a particular location inside the container
* Step2: Build Docker Image
  + The Dockerfile is used by the Docker CLI *build* command. The *build* command executes each instruction within the Dockerfile. The result is a built Docker Image that can be launched and run your configured app.
  + The build command takes in some different parameters. The format is *docker build -t <build-directory>*. The *-t* parameter allows you to specify a friendly name for the image and a tag, commonly used as a version number. This allows you to track built images and be confident about which version is being started.
  + Command:
    - docker build -t webserver-image:v1
    - To list all images
      * docker images
* Step3 Run:
  + The built Image can be launched in a consistent way to other Docker Images. When a container launches, it's sandboxed from other processes and networks on the host. When starting a container you need to give it permission and access to what it requires.
  + For example, to open and bind to a network port on the host you need to provide the parameter *-p <host-port>:<container-port>*.
  + Launch our newly built image providing the friendly name and tag. As it's a web server, bind port 80 to our host using the *-p* parameter.
    - docker run -d -p 80:80 webserver-image:v1
    - To access result
      * Curl docker

**Building Container Images:**

* Docker images are built based on a Dockerfile. A Dockerfile defines all the steps required to create a Docker image with your application configured and ready to be run as a container. The image itself contains everything, from operating system to dependencies and configuration required to run your application.
* Having everything within the image allows you to migrate images between different environments and be confident that if it works in one environment, then it will work in another.
* The Dockerfile allows for images to be composable, enabling users to extend existing images instead of building from scratch. By building on an existing image, you only need to define the steps to set up your application. The base images can be basic operating system installations or configured systems which simply need some additional customisations.
* Step1 Base Images
  + All Docker images start from a base image. A base image is the same images from the Docker Registry which are used to start containers. Along with the image name, we can also include the image tag to indicate which particular version we want, by default, this is the latest.
  + These base images are used as the foundation for your additional changes to run your application. For example, in this scenario, we require NGINX to be configured and running on the system before we can deploy our static HTML files. As such we want to use NGINX as our base image.
  + Dockerfile are simple text files with a command on each line. To define a base image we use the instruction *FROM <image-name>:<tag>*
  + Creating a DockerFile
    - he first line of the Dockerfile should be *FROM nginx:1.11-alpine*
    - FROM nginx:1.11-alpine
    - Caution
      * It's tempting to use the tag *:latest* however this can result in you building your image against a version which you were not expecting. We recommend that you always use a particular version number as your tag and manage the updating yourself.
* Step2 Running Commands
  + With the base image defined, we need to run various commands to configure our image. There are many commands to help with this, the main commands two are *COPY* and *RUN*.
  + *RUN <command>* allows you to execute any command as you would at a command prompt, for example installing different application packages or running a build command. The results of the RUN are persisted to the image so it's important not to leave any unnecessary or temporary files on the disk as these will be included in the image.
  + *COPY <src> <dest>* allows you to copy files from the directory containing the Dockerfile to the container's image. This is extremely useful for source code and assets that you want to be deployed inside your container.
  + A new *index.html* file has been created for you which we want to serve from our container. On the next line after the FROM command, use the COPY command to copy index.html into a directory called /usr/share/nginx/html
    - COPY index.html /usr/share/nginx/html/index.html
  + Protrip: If you're copying a file into a directory then you need to specify the filename as part of the destination.
* Step3 Exposing Ports
  + With our files copied into our image and any dependencies downloaded, you need to define which port application needs to be accessible on.
  + Using the *EXPOSE <port>* command you tell Docker which ports should be open and can be bound to. You can define multiple ports on the single command, for example, *EXPOSE 80 433* or *EXPOSE 7000-8000*
  + EXPOSE 80
* Step4 Default Commands
  + With the Docker image configured and having defined which ports we want accessible, we now need to define the command that launches the application.
  + The *CMD* line in a Dockerfile defines the default command to run when a container is launched. If the command requires arguments then it's recommended to use an array, for example ["cmd", "-a", "arga value", "-b", "argb-value"], which will be combined together and the command cmd -a "arga value" -b argb-value would be run.
  + The command to run NGINX is nginx -g daemon off;. Set this as the default command in the Dockerfile.
    - CMD ["nginx", "-g", "daemon off;"]
  + An alternative approach to *CMD* is *ENTRYPOINT*. While a CMD can be overridden when the container starts, an ENTRYPOINT defines a command which can have arguments passed to it when the container launches.
  + In this example, NGINX would be the entrypoint with -g daemon off; the default command.
  + Docker File:
    - FROM nginx:1.11-alpine
    - COPY index.html /usr/share/nginx/html/index.html
    - EXPOSE 80
    - CMD ["nginx", "-g", "daemon off;"]
* Step5 Building Containers
  + After writing your Dockerfile you need to use docker build to turn it into an image. The *build* command takes in a directory containing the Dockerfile, executes the steps and stores the image in your local Docker Engine. If one fails because of an error then the build stops.
  + Using the docker build command to build the image. You can give the image a friendly name by using the *-t <name>* option.
* Step6 Launching New Image
  + With the image successfully created, you can now launch the container in the same way we described in the first scenario.
  + Launch an instance of your newly built image using either the ID result from the build command or the friendly name you assigned it.
  + NGINX is designed to run as a background service so you should include the option *-d*. To make the web server accessible, bind it to port 80 using *p 80:80*
  + docker run -d -p 80:80 <image-id|friendly-tag-name>

**Dockerizing Node.js:**

* Step1 Base Image
  + As we described in the previous scenario, all images started with a base image, ideally as close to your desired configuration as possible. Node.js has pre-built images available with tags for each released version.
  + The image for Node 10.0 is node:10-alpine. This is an Alpine-based build which is smaller and more streamlined than the official image.
  + Alongside the base image, we also need to create the base directories of where the application runs from. Using the *RUN <command>* we can execute commands as if they're running from a command shell, by using mkdir we can create the directories where the application will execute from. In this case, an ideal directory would be */src/app* as the environment user has read/write access to this directory.
  + We can define a working directory using *WORKDIR <directory>* to ensure that all future commands are executed from the directory relative to our application.
  + Docker FIle:
    - FROM node:10-alpine
    - RUN mkdir -p /src/app
    - WORKDIR /src/app
* Step2 NPM install
  + In the previous set, we configured the foundation of our configuration and how we want the application to be deployed. The next stage is to install the dependencies required to run the application. For Node.js this means running NPM install.
  + To keep build times to a minimum, Docker caches the results of executing a line in the Dockerfile for use in a future build. If something has changed, then Docker will invalidate the current and all following lines to ensure everything is up-to-date.
  + With NPM we only want to re-run npm install if something within our *package.json* file has changed. If nothing has changed then we can use the cache version to speed up deployment. By using *COPY package.json <dest>* we can cause the *RUN npm install* command to be invalidated if the *package.json* file has changed. If the file has not changed, then the cache will not be invalidated, and the cached results of the npm install command will be used.
  + Docker File

FROM node:10-alpine

RUN mkdir -p /src/app

WORKDIR /src/app

**COPY package.json /src/app/package.json**

**RUN npm install**

* Step3 Configuring Application
  + After we've installed our dependencies, we want to copy over the rest of our application's source code. Splitting the installation of the dependencies and copying out source code enables us to use the cache when required.
  + If we copied our code before running *npm install* then it would run every time as our code would have changed. By copying just package.json we can be sure that the cache is invalidated only when our package contents have changed.
  + Create the desired steps in the Dockerfile to finish the deployment of the application.
  + We can copy the entire directory where our Dockerfile is using *COPY . <dest dir>*.
  + Once the source code has been copied, the ports the application requires to be accessed is defined using *EXPOSE <port>*.
  + Finally, the application needs to be started. One neat trick when using Node.js is to use the *npm start* command. This looks in the *package.json* file to know how to launch the application saving duplication of commands.

Docker File

FROM node:10-alpine

RUN mkdir -p /src/app

WORKDIR /src/app

COPY package.json /src/app/package.json

RUN npm install

**COPY . /src/app**

**EXPOSE 3000**

**CMD [ "npm", "start" ]**

* Step4 Building & Launching Container
  + docker build -t my-nodejs-app
  + docker run -d --name my-running-app -p 3000:3000 my-nodejs-app
  + curl <http://docker:3000>
* Step5 Environment Variables
  + Docker images should be designed so that they can be transferred from one environment to the other without making any changes or requiring to be rebuilt. By following this pattern you can be confident that if it works in one environment, such as staging, then it will work in another, such as production.
  + With Docker, environment variables can be defined when you launch the container. For example with Node.js applications, you should define an environment variable for *NODE\_ENV* when running in production.
  + Using *-e* option, you can set the name and value as *-e NODE\_ENV=production*
  + docker run -d --name my-production-running-app -e NODE\_ENV=production -p 3000:3000 my-nodejs-app

**Optimised Builds with Docker OnBuild**

* Step1 OnBuild
  + While Dockerfile are executed in order from top to bottom, you can trigger an instruction to be executed at a later time when the image is used as the base for another image.
  + The result is you can delay your execution to be dependent on the application which you're building, for example the application's *package.json* file.

**Docker FIle:**

FROM node:7

RUN mkdir -p /usr/src/app

WORKDIR /usr/src/app

ONBUILD COPY package.json /usr/src/app/

ONBUILD RUN npm install

ONBUILD COPY . /usr/src/app

CMD [ "npm", "start" ]

* Step2 Application Dockerfile
  + With all of the logic to copy the code, install our dependencies and launch our application the only aspect which needs to be defined on the application level is which port(s) to expose.
  + The advantage of creating OnBuild images is that our Dockerfile is now much simpler and can be easily reused across multiple projects without having to re-run the same steps improving build times.
    - FROM node:7-onbuild
    - EXPOSE 3000
* Step3 Building & Launching
  + The Dockerfile from the previous step has been created for you. Building the images based on the OnBuild docker file is the same as before. The OnBuild commands will be executed as if they were in the base Dockerfile.
  + docker build -t my-nodejs-ap
  + docker run -d --name my-running-app -p 3000:3000 my-nodejs-app
  + curl <http://docker:3000>

**Ignoring Files During Build**

* Step1 Docker ignore
  + To prevent sensitive files or directories from being included by mistake in images, you can add a file named *.dockerignore*.
  + A Dockerfile copies the working directory into the Docker Image. As a result, this would include potentially sensitive information such as a passwords file which we'd want to manage outside the image. View the Dockerfile with cat Dockerfile
  + Build the image
    - docker build -t password
  + Look at the output using
    - docker run password ls /app
  + Ignore File
    - The following command would include passwords.txt in our *.dockerignore* file and ensure that it didn't accidentally end up in a container. The *.dockerignore* file would be stored in source control and share with the team to ensure that everyone is consistent
    - echo passwords.txt >> .dockerignore
    - The ignore file supports directories and Regular expressions to define the restrictions, very similar to *.gitignore*. This file can also be used to improve build times which we'll investigate in the next step.
    - Build the image, because of the Docker Ignore file it shouldn't include the passwords file.
    - docker build -t nopassword
    - docker run nopassword ls /app
* Step2 Docker Build Context
  + The *.dockerignore* file can ensure that sensitive details are not included in a Docker Image. However they can also be used to improve the build time of images.
  + In the environment, a 100M temporary file has been created. This file is never used by the *Dockerfile*. When you execute a build command, Docker sends the entire path contents to the Engine for it to calculate which files to include. As a result sending the 100M file is unrequired and creates a slower build.
  + You can see the 100M impact by executing following the command.
  + Example: docker build -t large-file-context
* Step3 Optimised Build
  + In the same way, we used the *.dockerignore* file to exclude sensitive files, we can use it to exclude files which we don't want to be sent to the Docker Build Context during the build.
  + To speed up our build, simply include the filename of the large file in the ignore file.
  + echo big-temp-file.img >> .dockerignore
  + When we rebuild the image, it will be much faster as it doesn't have to copy the 100M file.
  + docker build -t no-large-file-context

**Create Data Containers**

* There are two ways of approaching stateful Containers, that is containers are store and persistent data for future use. This could be the container creating and storing data, for example, a database. Alternatively, it could be data requiring additional for instance the configuration or SSL certifications. This approach can also be used to backup data or debug containers.
* One approach we've discussed is using the *-v <host-dir>:<container-dir>* option to map directories.
* Step1 Create Container
  + Data Containers are containers whose sole responsibility is to be a place to store/manage data.
  + Like other containers they are managed by the host system. However, they don't run when you perform a docker ps command.
  + To create a Data Container we first create a container with a well-known name for future reference. We use busybox as the base as it's small and lightweight in case we want to explore and move the container to another host.
  + When creating the container, we also provide a *-v* option to define where other containers will be reading/saving data.
  + docker create -v /config --name dataContainer busybox
* Step2 Copy Files
  + With the container in place, we can now copy files from our local client directory into the container.
  + To copy files into a container you use the command *docker cp*. The following command will copy the config.conf file into our *dataContainer* and the directory *config*.
  + docker cp config.conf dataContainer:/config/
* Step3 Mount Volumes From
  + Now our Data Container has our config, we can reference the container when we launch dependent containers requiring the configuration file.
  + Using the *--volumes-from <container>* option we can use the mount volumes from other containers inside the container being launched. In this case, we'll launch an Ubuntu container which has reference to our Data Container. When we list the config directory, it will show the files from the attached container.
  + docker run --volumes-from dataContainer ubuntu ls /config
  + If a */config* directory already existed then, the volumes-from would override and be the directory used. You can map multiple volumes to a container.
* Step4 Export/Import Containers
  + If we wanted to move the Data Container to another machine then we can export it to a .tar file.
  + docker export dataContainer > dataContainer.tar
  + The command docker import dataContainer.tar will import the Data Container back into Docker

**Creating networks Between Containers using Links.**

* Step1 Start Redis
  + The most common scenario for connecting to containers is an application connecting to a data-store. The key aspect when creating a link is the name of the container. All containers have names, but to make it easier when working with links, it's important to define a friendly name for the source container which you're connecting to.
  + Run a redis server with a friendly name of *redis-server* which we'll connect to in the next step. This will be our source container.
  + docker run -d --name redis-server redis
* Step2 Create Link
  + To connect to a source container you use the *--link <container-name|id>:<alias>* option when launching a new container. The container name refers to the source container we defined in the previous step while the alias defines the friendly name of the host.
  + By setting an alias we separate how our application is configured to how the infrastructure is called. This means the application configuration doesn't need to change as it's connected to other environments.
  + In this example, we bring up a Alpine container which is linked to our *redis-server*. We've defined the alias as *redis*. When a link is created, Docker will do two things.
  + First, Docker will set some environment variables based on the linked to the container. These environment variables give you a way to reference information such as Ports and IP addresses via known names.
  + docker run --link redis-server:redis alpine env
  + Secondly, Docker will update the *HOSTS* file of the container with an entry for our source container with three names, the original, the alias and the hash-id. You can output the containers host entry using cat /etc/hosts
  + docker run --link redis-server:redis alpine cat /etc/hosts
  + With a link created you can ping the source container in the same way as if it were a server running in your network.
  + docker run --link redis-server:redis alpine ping -c 1 redis
* Step3 Connect To App
  + With a link created, applications can connect and communicate with the source container in the usual way, independent of the fact both services are running in containers.
  + Here is a simple node.js application which connects to redis using the hostname *redis*.
  + docker run -d -p 3000:3000 --link redis-server:redis katacoda/redis-node-docker-example
  + Sending an HTTP request to the application will store the request in Redis and return a count. If you issue multiple requests, you'll see the counter increment as items are persisted.
  + url docker:3000
* Step4 Connect to Redis CLI
  + In the same way, you can connect source containers to applications, you can also connect them to their own CLI tools.
  + The command below will launch an instance of the Redis-cli tool and connect to the redis server via its alias.
  + docker run -it --link redis-server:redis redis redis-cli -h redis
  + The command KEYS \* will output the contents stored currently in the source redis container.

**Creating Networks Between Containers using Networks**

* Docker has two approaches to networking. The first defines a link between two containers. This link updates */etc/hosts* and environment variables to allow containers to discover and communicate.
* The alternate approach is to create a *docker network* that containers are connected to. The network has similar attributes to a physical network, allowing containers to come and go more freely than when using links.
* Step1 Create Network
  + The first step is to create a network using the CLI. This network will allow us to attach multiple containers which will be able to discover each other.
  + In this example, we're going to start by creating a *backend-network*. All containers attached to our backend will be on this network.
  + Create Network
    - In this example, we're going to start by creating a *backend-network*. All containers attached to our backend will be on this network.
  + Connect to Network
    - When we launch new containers, we can use the *--net* attribute to assign which network they should be connected to.
    - docker run -d --name=redis --net=backend-network redis
* Step2 Network Communication
  + Unlike using links, *docker networks* behave like traditional networks where nodes can be attached/detached.
  + The first thing you'll notice is that Docker no longer assigns environment variables or updates the hosts file of containers. Explore using the following two commands and you'll notice it no longer mentions other containers.
    - docker run --net=backend-network alpine env
    - docker run --net=backend-network alpine cat /etc/hosts
  + Instead, the way containers can communicate via an Embedded DNS Server in Docker. This DNS server is assigned to all containers via the IP 127.0.0.11 and set in the *resolv.conf* file.
    - docker run --net=backend-network alpine cat /etc/resolv.conf
  + When containers attempt to access other containers via a well-known name, such as Redis, the DNS server will return the IP address of the correct Container. In this case, the fully qualified name of Redis will be *redis.backend-network*.
    - docker run --net=backend-network alpine ping -c1 redis
* Step3 Connect Two Containers
  + Docker supports multiple networks and containers being attached to more than one network at a time.
  + For example, let's create a separate network with a Node.js application that communicates with our existing Redis instance.
  + The first task is to create a new network in the same way.
    - docker network create frontend-network
  + When using the *connect* command it is possible to attach existing containers to the network.
    - docker network connect frontend-network redis
  + When we launch the web server, given it's attached to the same network it will be able to communicate with our Redis instance.
    - docker run -d -p 3000:3000 --net=frontend-network katacoda/redis-node-docker-example
* Step4 Create Aliases
  + Links are still supported when using *docker network* and provide a way to define an Alias to the container name. This will give the container an extra DNS entry name and way to be discovered. When using --link the embedded DNS will guarantee that localised lookup result only on that container where the --link is used.
  + The other approach is to provide an alias when connecting a container to a network.
  + The following command will connect our Redis instance to the frontend-network with the alias of *db*.
    - docker network create frontend-network2
    - docker network connect --alias db frontend-network2 redis
  + When containers attempt to access a service via the name db, they will be given the IP address of our Redis container.
    - docker run --net=frontend-network2 alpine ping -c1 db
* Step5 Disconnect Containers
  + With our networks created, we can use the CLI to explore the details.
  + Command to list all the networks on host.
    - docker network ls
  + We can then explore the network to see which containers are attached and their IP addresses.
    - docker network inspect frontend-network
  + Command disconnects the redis container from the *frontend-network*.
    - docker network disconnect frontend-network redis

**Persisting Data Using Volumes**

* Step1 Data Volumes
  + Docker Volumes are created and assigned when containers are started. Data Volumes allow you to map a host directory to a container for sharing data.
  + This mapping is bi-directional. It allows data stored on the host to be accessed from within the container. It also means data saved by the process inside the container is persisted on the host
  + This example will use Redis as a way to persist data. Start a Redis container below, and create a data volume using the *-v* parameter. This specifies that any data saved inside the container to the */data* directory should be persisted on the host in the directory */docker/redis-data*.
    - docker run -v /docker/redis-data:/data \
    - > --name r1 -d redis \
    - > redis-server --appendonly yes
  + We can pipe data into the Redis instance using the following command.
    - cat data | docker exec -i r1 redis-cli --pipe
  + Redis will save this data to disk. On the host we can investigate the mapped direct which should contain the Redis data file.
    - ls /docker/redis-data
  + This same directory can be mounted to a second container. One usage is to have a Docker Container performing backup operations on your data.
    - docker run -v /docker/redis-data:/backup ubuntu ls /backup
* Step2 Shared Volumes
  + Data Volumes mapped to the host are great for persisting data. However, to gain access to them from another container you need to know the exact path which can make it error-prone.
  + An alternate approach is to use *-volumes-from*. The parameter maps the mapped volumes from the source container to the container being launched
  + In this case, we're mapping our Redis container's volume to an Ubuntu container. The */data* directory only exists within our Redis container, however, because *-volumes-from* our Ubuntu container can access the data.
    - docker run --volumes-from r1 -it ubuntu ls /data
  + This allows us to access volumes from other containers without having to be concerned how they're persisted on the host.
* Step3 Read-Only Volumes
  + Mounting Volumes gives the container full read and write access to the directory. You can specify read-only permissions on the directory by adding the permissions *:ro* to the mount.
  + If the container attempts to modify data within the directory it will error.
  + docker run -v /docker/redis-data:/data:ro -it ubuntu rm -rf /data

**Manage Container Log FIles**

* Step1 Docker Logs
  + When you start a container, Docker will track the Standard Out and Standard Error outputs from the process and make them available via the client.
  + Example:In the background, there is an instance of Redis running with the name *redis-server*. Using the Docker client, we can access the standard out and standard error outputs using docker logs redis-server
* Step2 SysLog
  + By default, the Docker logs are outputted using the *json-file* logger meaning the output is stored in a JSON file on the host. This can result in large files filling the disk. As a result, you can change the log driver to move to a different destination.
  + The Syslog log driver will write all the container logs to the central syslog on the host. "syslog is a widely used standard for message logging. It permits separation of the software that generates messages, the system that stores them, and the software that reports and analyses them."
  + This log-driver is designed to be used when syslog is being collected and aggregated by an external system.
  + Example: The command below will redirect the redis logs to syslog.
    - docker run -d --name redis-syslog --log-driver=syslog redis
  + If you attempt to view the logs using the client you'll receive the error *FATA[0000] "logs" command is supported only for "json-file" logging driver*, Instead, you need to access them via the syslog stream.
* Step3 Disable Logging
  + The third option is to disable logging on the container. This is particularly useful for containers which are very verbose in their logging.
  + When the container is launched simply set the log-driver to none. No output will be logged.
    - docker run -d --name redis-none --log-driver=none redis
  + The *inspect* command allows you to identify the logging configuration for a particular container. The command below will output the LogConfig section for each of the containers.
    - Server created in step 1
      * docker inspect --format '{{ .HostConfig.LogConfig }}' redis-server
    - Server created in step 2
      * docker inspect --format '{{ .HostConfig.LogConfig }}' redis-syslog
    - Server created in this step
      * docker inspect --format '{{ .HostConfig.LogConfig }}' redis-none

**Ensuring Container Uptime With Restart Policies**

* This scenario will explore how you can keep containers alive and automatically restart them if they crash unexpectedly.
* Step1 Stop On Fail
  + Docker considers any containers to exit with a non-zero exit code to have crashed. By default a crashed container will remain stopped.
  + Example: We've created a special container which outputs a message and then exits with code 1 to simulate a crash.
    - docker run -d --name restart-default scrapbook/docker-restart-example
  + If you list all the containers, including stopped, you will see the container has crashed
    - docker ps -a
  + While the logs will output our message, which in real-life would hopefully indicate information to help us diagnose the issue.
    - docker logs restart-default
* Step2 Restart On Fail
  + Depending on your scenario, restarting a failed process might correct the problem. Docker can automatically retry to launch the Docker a specific number of times before it stops trying.
  + Example: The option *--restart=on-failure:#* allows you to say how many times Docker should try again. In the example below, Docker will restart the container three times before stopping.
    - docker run -d --name restart-3 --restart=on-failure:3 scrapbook/docker-restart-example
  + As we can see from the logs, it was launched on three occasions.
    - docker logs restart-3
* Step3 Always Restart
  + Finally Docker can always restart a failed container, in this case, Docker will keep trying until the container it is explicitly told to stop.
  + Example: Use the *always* flag to automatically restart the container when is crashes for example
    - docker run -d --name restart-always --restart=always scrapbook/docker-restart-example
  + You can view the restart attempting via the log
    - docker logs restart-always

**Adding Docker Metadata & Labels**

* When running containers in production, it can be useful to add additional metadata relating to the container to help their management. This metadata could be related to which version of the code is running, which applications or users own the container or define special criteria such as which servers they should run on.
* This additional data is managed via Docker Labels. Labels allow you to define custom metadata about a container or image which can later be inspected or used as part of a filter.
* Step1 Docker Containers
  + Labels can be attached to containers when they are launched via docker run. A container can have multiple labels attached to them at any one time.
  + Single Label
    - To add a single label you use the *l =<value>* option. The example below assigns a label called user with an ID to the container. This would allow us to query for all the containers running related to that particular user.
      * docker run -l user=12345 -d redis
  + External File
    - If you're adding multiple labels, then these can come from an external file. The file needs to have a label on each line, and then these will be attached to the running container.
    - This line creates two labels in the file, one for the user and the second assigning a role.
    - echo 'user=123461' >> labels && echo 'role=cache' >> labels
    - The *--label-file=<filename>* option will create a label for each line in the file.
    - docker run --label-file=labels -d redis
* Step2 Docker Images
  + Labelling images work in the same way as containers but are set in the *Dockerfile* when the image is built. When a container has launched the labels of the image will be applied to the container instance.
  + Single Label
    - Within a *Dockerfile* you can assign a label using the LABEL instruction. Below the label *vendor* is created with the name Scrapbook.
    - LABEL vendor=Katacoda
  + Multiple Labels
    - If we want to assign multiple labels then, we can use the format below with a label on each line, joined using a back-slash ("\"). Notice we're using the DNS notation format for labels which are related to third party tooling.
    - LABEL vendor=Katacoda \ com.katacoda.version=0.0.5 \ com.katacoda.build-date=2016-07-01T10:47:29Z \ com.katacoda.course=Docker
* Step3 Inspect
  + Labels and Metadata are only useful if you can view/query them later. The first approach to viewing all the labels for a particular container or image is by using docker inspect.
  + By providing the running container's friendly name or hash id, you can query all of it's metadata.
    - docker inspect rd
  + Using the *-f* option you can filter the JSON response to just the Labels section we're interested in.
    - docker inspect -f "{{json .Config.Labels }}" rd
  + Inspecting images works in the same way however the JSON format is slightly different, naming it *ContainerConfig* instead of *Config*.
    - docker inspect -f "{{json .ContainerConfig.Labels }}" katacoda-label-example
  + These labels will remain even if the image has been untagged. When an image is untagged, it will have the name *<none>*
* Step4 Query By Label
  + While inspecting individual containers and images provides you with more context, on a production running potentially thousands of containers, it's useful to limit the responses to the containers you're interested in.
  + Filtering Containers
    - The docker ps command allows you to specify a filter based on a label name and value. For example, the query below will return all the containers which have a *user* label key with the value *katacoda*.
    - docker ps --filter "label=user=scrapbook"
  + Filtering Images
    - The same filter approach can be applied to images based on the labels used when the image was built.
    - docker images --filter "label=vendor=Katacoda"
  + When querying both the label key name and value are case sensitive.
* Step5 Daemon Labels
  + Labels are not only applied to images and containers but also the Docker Daemon itself. When you launch an instance of the daemon, you can assign it labels to help identify how it should be used, for example, if it's a development or production server or if it's more suited to particular roles such running databases.

docker -d \

-H unix:///var/run/docker.sock \

--label com.katacoda.environment="production" \

--label com.katacoda.storage="ssd"

**Load Balancing Containers**

* With Docker, there are two main ways for containers to communicate with each other. The first is via links which configure the container with environment variables and host entry allowing them to communicate. The second is using the Service Discovery pattern which uses information provided by third parties.
* The Service Discovery pattern is where the application uses a third party system to identify the location of the target service. For example, if our application wanted to talk to a database, it would first ask an API what the IP address of the database is. This pattern allows you to quickly reconfigure and scale your architectures with improved fault tolerance than fixed locations.
* Step1 NGINX proxy
  + In this scenario, we want to have a NGINX service running which can dynamically discover and update its load balance configuration when new containers are loaded. Thankfully this has already been created and is called [nginx-proxy](https://github.com/jwilder/nginx-proxy).
  + Nginx-proxy accepts HTTP requests and proxies the request to the appropriate container based on the request Hostname. This is transparent to the user which happens without any additional performance overhead.
  + There are three key properties required to be configured when launching the proxy container.
    - The first is binding the container to port 80 on the host using *-p 80:80*. This ensures all HTTP requests are handled by the proxy
    - The second is to mount the *docker.sock* file. This is a connection to the Docker daemon running on the host and allows containers to access its metadata via the API. Nginx-proxy uses this to listen for events and then updates the NGINX configuration based on the container IP address. Mounting file works in the same way as directories using *-v /var/run/docker.sock:/tmp/docker.sock:ro*. We specify *:ro* to restrict access to read-only.
    - Finally, we can set an optional \_-e DEFAULT*HOST=<domain>*. If a request comes in and doesn't make any specified hosts, then this is the container where the request will be handled. This enables you to run multiple websites with different domains on a single machine with a fall-back to a known website.
  + Use the command below to launch nginx-proxy.
    - docker run -d -p 80:80 -e DEFAULT\_HOST=proxy.example -v /var/run/docker.sock:/tmp/docker.sock:ro --name nginx jwilder/nginx-proxy
  + Because we're using a DEFAULT\_HOST, any requests which come in will be directed to the container that has been assigned the HOST proxy.example.
  + You can make a request to the web server using below command
    - curl <http://docker>
  + As we have no containers, it will return a 503 error.
* Step2 Single Host
  + Nginx-proxy is now listening to events which Docker raises on start / stop. A sample website called *katacoda/docker-http-server* has been created which returns the machine name its running on. This allows us to test that our proxy is working as expected. Internally its a PHP and Apache2 application listening on port 80.
  + For Nginx-proxy to start sending requests to a container you need to specify the *VIRTUAL\_HOST* environment variable. This variable defines the domain where requests will come from and should be handled by the container.
    - docker run -d -p 80 -e VIRTUAL\_HOST=proxy.example katacoda/docker-http-server
  + curl http://docker

<h1>This request was processed by host: de380507f74f</h1>

* Step3 Cluster
  + We now have successfully created a container to handle our HTTP requests. If we launch a second container with the same VIRTUAL\_HOST then nginx-proxy will configure the system in a round-robin load balanced scenario. This means that the first request will go to one container, the second request to a second container and then repeat in a circle. There is no limit to the number of nodes you can have running.
  + Launch a second container using the same command as we did before.
    - docker run -d -p 80 -e VIRTUAL\_HOST=proxy.example katacoda/docker-http-server
  + If we execute a request to our proxy using curl http://docker then the request will be handled by our first container. A second HTTP request will return a different machine name meaning it was dealt with by our second container.

$ curl http://docker

<h1>This request was processed by host: 6e21516168d3</h1>

$ curl http://docker

<h1>This request was processed by host: de380507f74f</h1>

$ curl http://docker

<h1>This request was processed by host: 6e21516168d3</h1>

$ curl http://docker

<h1>This request was processed by host: de380507f74f</h1>

* Step4 Generated NGINX Configuration
  + While *nginx-proxy* automatically creates and configures NGINX for us, if you're interested in what the final configuration looks like then you can output the complete config file with *docker exec* as shown below.
    - docker exec nginx cat /etc/nginx/conf.d/default.conf
  + Additional information about when it reloads configuration can be found in the logs using
    - docker logs nginx

**Orchestration using Docker Compose**

* When working with multiple containers, it can be difficult to manage the starting along with the configuration of variables and links. To solve this problem, Docker has a tool called Docker Compose to manage the orchestration, of launching, of containers.
* Step1 Defining First Container
  + Docker Compose is based on a *docker-compose.yml* file. This file defines all of the containers and settings you need to launch your set of clusters. The properties map onto how you use the docker run commands, however, are now stored in source control and shared along with your code.
  + The format of the file is based on YAML (Yet Another Markup Language).

container\_name:

property: value

- or options

* + In this scenario, we have a Node.js application which requires connecting to Redis. To start, we need to define our docker-compose.yml file to launch the Node.js application.
  + Given the format above, the file needs to name the container 'web' and set the build property to the current directory. We'll cover the other properties in future steps.

web:

build: .

* Step2 Defining Settings
  + Docker Compose supports all of the properties which can be defined using docker run.
  + To link two containers together to specify a links property and list required connections. For example, the following would link to the redis source container defined in the same file and assign the same name to the alias.

links:

- redis

* + The same format is used for other properties such as ports

ports:

- "3000"

- "8000

* + For more information
    - <https://docs.docker.com/compose/compose-file/>
* Step3 Defining Second Container
  + In the previous step, we used the Dockerfile in the current directory as the base for our container. In this step, we want to use an existing image from Docker Hub as a second container.
  + To find the second container you simply use the same format as before on a new line. The YAML format is flexible enough to define multiple containers within the same file.
  + Define the second container with the name redis which uses the image redis. Following the YAML format, the container details would be:

redis:

image: redis:alpine

volumes:

- /var/redis/data:/data

* + Docker compose file now.

web:

build: .

links:

- redis

ports:

- "3000"

- "8000"

redis:

image: redis:alpine

volumes:

- /var/redis/data:/data

* Step4 Docker up
  + With the created docker-compose.yml file in place, you can launch all the applications with a single command of up. If you want to bring up a single container, then you can use up <name>.
  + The *-d* argument states to run the containers in the background, similar to when used with docker run.
  + Command:
    - docker-compose up -d
* Step5 Docker Management
  + Not only can Docker Compose manage starting containers but it also provides a way to manage all the containers using a single command.
  + For example, to see the details of the launched containers you can use
    - docker-compose ps
  + To access all the logs via a single stream you use
    - docker-compose logs
  + Other commands follow the same pattern. Discover them by typing
    - Docker-compose
* Step6 Docker Scale
  + As Docker Compose understands how to launch your application containers, it can also be used to scale the number of containers running.
  + The scale option allows you to specify the service and then the number of instances you want. If the number is greater than the instances already running then, it will launch additional containers. If the number is less, then it will stop the unrequired containers.
  + Scale the number of web containers you're running using the command
    - docker-compose scale web=3
  + You can scale it back down using
    - docker-compose scale web=1
* Step7 Docker Stop
  + As when we launched the application, to stop a set of containers you can use the command
    - docker-compose stop
  + To remove all the containers use the command
    - docker-compose rm

**See Container Metrics with Docker Stats**

* When running containers in production, it's important to monitor there runtime metrics, such as CPU usage and memory, to ensure they're behaving as expected. These metrics can also help diagnose issues if they occur.
* Step1 Single Container
  + Command
    - docker stats nginx
* Step2 Multiple Containers
  + The built-in Docker allows you to provide multiple names/ids and display their stats within a single window.
  + Command:
    - docker ps -q | xargs docker stats

**Creating Optimised Docker Images using Multi-Stage Builds**

* The feature is ideal for deploying languages such as Golang as containers. By having multi-stage builds, the first stage can build the Golang binary using a larger Docker image as the base. In the second stage, the newly built binary can be deployed using a much smaller base image. The end result is an optimised Docker Image.
* Step1 DockerFile
  + The Multi-Stage feature allows a single Dockerfile to contain multiple stages in order to produce the desired, optimised, Docker Image.
  + Previously, the problem would have been solved with two Dockerfiles. One file would have the steps to build the binary and artifacts using a development container, the second would be optimised for production and not include the development tools.
  + By removing development tooling in the production image, you reproduce the attack surface and improve the deployment time.
  + Start by deploying a sample [Golang HTTP Server](https://github.com/katacoda/golang-http-server). This currently using a two staged Docker Build approach. This scenario will create a new Dockerfile that allows the image to be built using a single command.
    - git clone <https://github.com/katacoda/golang-http-server.git>
  + Using the editor, create a Multi-Stage Dockerfile. The first stage using the Golang SDK to build a binary. The second stage copies the resulting binary into a optimised Docker Image.
    - File: Dockerfile.multi

# First Stage

FROM golang:1.6-alpine

RUN mkdir /app

ADD . /app/

WORKDIR /app

RUN CGO\_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .

# Second Stage

FROM alpine

EXPOSE 80

CMD ["/app"]

# Copy from first stage

COPY --from=0 /app/main /app

* Step2 Build Multi-Stage Docker Image
  + With the new syntax for the Dockerfile in place, the build process is identical to previously.
  + Create the desired Docker Image using the build command below.
    - docker build -f Dockerfile.multi -t golang-app
  + The result will be two images. One untagged that was used for the first stage and the second, smaller image, our target image.
    - docker images
  + If you receive the error, *COPY --from=0 /build/out /app/ Unknown flag: from*, it means you're running an older version of Docker without the multi-stage support. Step 1 of this scenario upgrades the current Docker version.
* Step3 Test Image
  + docker run -d -p 80:80 golang-app
  + curl localhost

**Formatting PS Output**

* Example1 Names and Images as Table
  + The format of docker ps can be formatted to only display the information relevant to you.
  + Start by launching a example container - docker run -d redis
  + The standard docker ps command outputs the name, image used, command, uptime and port information.
  + To limit which columns are displayed, use the \_--format\_\_ parameter. The parameter allows pretty-printing containers using a Go template syntax
    - docker ps --format '{{.Names}} container is using {{.Image}} image'
  + As it's using Go templates, it includes helper functions such as *tables*.

docker ps --format 'table {{.Names}}\t{{.Image}}'

NAMES IMAGE

sharp\_chandrasekhar redis

* Example2 List IP addresses
  + However, the format parameter supports displaying data that is already exposed via the docker ps command. If you want to include additional information, such as the IP Address of the container, then the data needs to come via docker inspect.
  + Thankfully, the docker inspect also supports pretty-printing the results via a Go Template. The container IDs from docker ps can be piped into docker inspect.
  + The format parameter can then access all of the container information. Below is an example of listing all the IP addresses for the running containers.
    - docker ps -q | xargs docker inspect --format '{{ .Id }} - {{ .Name }} - {{ .NetworkSettings.IPAddress }}'

**Run Docker From Rootless Users**

* With Docker all the containers are managed via the Docker Daemon. The Daemon controls all aspects of the container lifecycle.
* Previous versions of Docker required that the Daemon started by user with root privileges. This required giving users full access to a machine in order to control and configure Docker. As a result, this exposed potential security risks.
* Rootless Docker is a project from Docker that removes the requirement for the Docker Daemon to be started by a root. This creates a more secure environment.
* Step1 Create Ubuntu User
  + The useradd command will create a user with the default permissions. Run the command in the terminal to add a new user called lowprivuser. This user can be called anything.
    - useradd -m -d /home/lowprivuser -p $(openssl passwd -1 password) lowprivuser
  + Switch user
    - sudo su lowprivuser
  + When running as this user, a couple of items change. For example, the user is not able to create or change files in certain locations such as the root directory

lowprivuser@host01:/root$ touch /root/blocked

touch: cannot touch '/root/blocked': Permission denied

* + The user is also not able to access Docker as previously this required them to have root permissions.

lowprivuser@host01:/root$ docker ps

Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Get http://%2Fvar%2Frun%2Fdocker.sock/v1.40/containers/json: dial unix /var/run/docker.sock: connect: permission denied

* Step2 Install Rootless Docker
  + Docker has made available a script which will deploy the required components for the new Rootless version.
  + Run the following command as *lowprivuser* to execute the script and install the components.
    - curl -sSL https://get.docker.com/rootless | sh
* Step3 Access Docker
  + Rootless Docker has now been installed. The daemon can be started using the following script.

export XDG\_RUNTIME\_DIR=/tmp/docker-1001

export PATH=/home/lowprivuser/bin:$PATH

export DOCKER\_HOST=unix:///tmp/docker-1001/docker.sock

mkdir -p $XDG\_RUNTIME\_DIR

/home/lowprivuser/bin/dockerd-rootless.sh --experimental --storage-driver vfs

* + This will run in the foreground and allow you to see the debug output from the Rootless Docker Daemon.
  + Click the following command to launch a second terminal window and change the context to run as roootlessuser.

sudo su lowprivuser

: "Second Terminal running as lowprivuser"

Id

* + To access Docker, set the following environment variables. This specifies connecting to the Docker instance running for the user with id *1001*, which should match the id of *lowprivuser*.

export XDG\_RUNTIME\_DIR=/tmp/docker-1001

export PATH=/home/lowprivuser/bin:$PATH

export DOCKER\_HOST=unix:///tmp/docker-1001/docker.sock

* Step4 Run Containers
  + It's now possible to access the Docker Daemon running for user 1001.
  + The standard Docker CLI commands work in the same way. The following command lists all the containers running for the user, currently it should return an empty list.
    - docker ps
  + It's possible to inspect details of the Daemon running:
    - docker info
  + Containers can be in the same way.
    - docker run -it ubuntu bash
  + Users within the container will still be reported as root. They will be able to install packages and modify parts of the system running inside of Docker. However, if they managed to break out they wouldn't be able to interfere with the host.
    - id
  + In a separate terminal window, as root it's possible to explore which processes are running and which user started them. Using *ps aux* you can verify that our new container instance is managed and owned by our low privileged user.
    - id; ps aux | grep lowprivuser
  + The system is now running Docker Containers without requiring any additional permissions, allowing our systems to operate with increased security.

**Swarm Mode:**

* In 1.12, Docker introduced Swarm Mode. Swarm Mode enables the ability to deploy containers across multiple Docker hosts, using overlay networks for service discovery with a built-in load balancer for scaling the services.
* Swarm Mode is managed as part of the Docker CLI, making it a seamless experience to the Docker ecosystem.
* Key Concepts
  + Node: A Node is an instance of the Docker Engine connected to the Swarm. Nodes are either managers or workers. Managers schedule which containers to run where. Workers execute the tasks. By default, Managers are also workers.
  + Services: A service is a high-level concept relating to a collection of tasks to be executed by workers. An example of a service is an HTTP Server running as a Docker Container on three nodes.
  + Load Balancing: Docker includes a load balancer to process requests across all containers in the service.
* Step1 Initialize Swarm MOde
  + Turn a single host Docker host into a Multi-host Docker Swarm Mode. Becomes Manager By default, Docker works as an isolated single-node. All containers are only deployed onto the engine. Swarm Mode turns it into a multi-host cluster-aware engine.
  + The first node to initialise the Swarm Mode becomes the manager. As new nodes join the cluster, they can adjust their roles between managers or workers. You should run 3-5 managers in a production environment to ensure high availability.
  + Swarm Mode is built into the Docker CLI. You can find an overview the possibility commands via docker swarm --help
  + The most important one is how to initialise Swarm Mode. Initialisation is done via *init*.
    - docker swarm init
  + After running the command, the Docker Engine knows how to work with a cluster and becomes the manager. The results of an initialisation is a token used to add additional nodes in a secure fashion. Keep this token safe and secure for future use when scaling your cluster.
* Step2 Join Cluster
  + With Swarm Mode enabled, it is possible to add additional nodes and issue commands across all of them. If nodes happen to disappear, for example, because of a crash, the containers which were running on those hosts will be automatically rescheduled onto other available nodes. The rescheduling ensures you do not lose capacity and provides high-availability.
  + On each additional node, you wish to add to the cluster, use the Docker CLI to join the existing group. Joining is done by pointing the other host to a current manager of the cluster. In this case, the first host.
  + Docker now uses an additional port, *2377*, for managing the Swarm. The port should be blocked from public access and only accessed by trusted users and nodes. We recommend using VPNs or private networks to secure access.
  + The first task is to obtain the token required to add a worker to the cluster. For demonstration purposes, we'll ask the manager what the token is via *swarm join-token*. In production, this token should be stored securely and only accessible by trusted individuals.
    - token=$(ssh -o StrictHostKeyChecking=no 172.17.0.73 "docker swarm join-token -q worker") && echo $toke
  + On the second host, join the cluster by requesting access via the manager. The token is provided as an additional parameter.
    - docker swarm join 172.17.0.73:2377 --token $token
  + By default, the manager will automatically accept new nodes being added to the cluster. You can view all nodes in the cluster using docker node ls
* Step3 Create Overlay Network
  + Swarm Mode also introduces an improved networking model. In previous versions, Docker required the use of an external key-value store, such as Consul, to ensure consistency across the network. The need for consensus and KV has now been incorporated internally into Docker and no longer depends on external services.
  + The improved networking approach follows the same syntax as previously. The *overlay* network is used to enable containers on different hosts to communicate. Under the covers, this is a Virtual Extensible LAN (VXLAN), designed for large scale cloud based deployments.
  + The following command will create a new overlay network called *skynet*. All containers registered to this network can communicate with each other, regardless of which node they are deployed onto.
    - docker network create -d overlay skynet
* Step4 Deploy Service
  + By default, Docker uses a spread replication model for deciding which containers should run on which hosts. The spread approach ensures that containers are deployed across the cluster evenly. This means that if one of the nodes is removed from the cluster, the instances would be already running on the other nodes. The workload on the removed node would be rescheduled across the remaining available nodes.
  + A new concept of Services is used to run containers across the cluster. This is a higher-level concept than containers. A service allows you to define how applications should be deployed at scale. By updating the service, Docker updates the container required in a managed way.
  + In this case, we are deploying the Docker Image *katacoda/docker-http-server*. We are defining a friendly name of a service called *http* and that it should be attached to the newly created *skynet* network.
  + For ensuring replication and availability, we are running two instances, of replicas, of the container across our cluster.
  + Finally, we load balance these two containers together on port *80*. Sending an HTTP request to any of the nodes in the cluster will process the request by one of the containers within the cluster. The node which accepted the request might not be the node where the container responds. Instead, Docker load-balances requests across all available containers.
    - docker service create --name http --network skynet --replicas 2 -p 80:80 katacoda/docker-http-server
  + You can view the services running on the cluster using the CLI command
    - docker service ls
  + As containers are started you will see them using the *ps* command. You should see one instance of the container on each host.
  + List containers on the first host - docker ps
  + List containers on the second host - docker ps
  + If we issue an HTTP request to the public port, it will be processed by the two containers curl host01
* Step5 Inspect State
  + The Service concept allows you to inspect the health and state of your cluster and the running applications.
  + We can view the list of all the tasks associated with a service across the cluster. In this case, each task is a container docker service ps http
  + You can view the details and configuration of a service via
    - docker service inspect --pretty http
  + On each node, you can ask what tasks it is currently running. Self refers to the manager node Leader: docker node ps self
  + Using the ID of a node you can query individual hosts
    - docker node ps $(docker node ls -q | head -n1)
* Step6 Scale Service
  + A Service allows us to scale how many instances of a task are running across the cluster. As it understands how to launch containers and which containers are running, it can easily start, or remove, containers as required. At the moment the scaling is manual. However, the API could be hooked up to an external system such as a metrics dashboard.
  + At present, we have two load-balanced containers running, which are processing our requests curl host01
  + The command below will scale our *http* service to be running across five containers.
    - docker service scale http=5
  + On each host, you will see additional nodes being started docker ps
  + The load balancer will automatically be updated. Requests will now be processed across the new containers. Try issuing more commands via
    - curl host01

**Commands:**

* To Launch process
  + docker run -d --name=db redis:alpine
* To view process
  + ps aux | grep redis-server
* To get information like PID (Process ID) and PPID (Parent Process ID)
  + docker top db
* To find parent process
  + ps aux | grep <ppid>
* To list all the sub process
  + pstree -c -p -A $(pgrep dockerd)
* To list namespace
  + docker run -d --name=web --net=container:db nginx:alpine
  + WEBPID=$(pgrep nginx | tail -n1)
  + echo nginx is $WEBPID
  + cat /proc/$WEBPID/cgroup
  + ls -lha /proc/$WEBPID/ns
  + ls -lha /proc/$WEBPID/ns/ | grep net