Getting Started with Docker

What is Docker?

Docker is a set of tools that allow for applications and services to be developed, built and deployed in isolation while allowing consistent operation regardless of the host platform.

What is Docker Not?

Docker is not a replacement for virtualization products like Parallels Desktop, VMWare or VirtualBox. Virtualization products allow you to simulate a complete desktop/server environment and is meant to be long lasting. Docker on the other hand is designed to provide a single function per docker container. The docker containers should not need to persist changes to data or newly created files across restarts. It is very common, especially in large deployment environments, to delete docker containers as soon as they have shut down.

Getting Started

Key vocabulary to get started (mostly from and more at https://docs.docker.com/glossary):

- Docker Desktop
  - Mac
  - Windows
  - an easy-to-install, lightweight Docker development environment
- container
  - A container is a runtime instance of a docker image.
- docker image
  - An Image is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. An image typically contains a union of layered filesystems stacked on top of each other. An image does not have state and it never changes
- Dockerfile
  - A Dockerfile is a text document that contains all the commands you would normally execute manually in order to build a Docker image. Docker can build images automatically by reading the instructions from a Dockerfile.
- registry
  - A Registry is a hosted service containing repositories of images which responds to the Registry API.
- repository
  - A repository is a set of Docker images. A repository can be shared by pushing it to a registry server. The different images in the repository can be labeled using tags.
- Port binding
  - Associating a running service or program with a network interface and port to allow for external connections

Install Docker Desktop

- Docker Desktop can be downloaded from https://www.docker.com/products/docker-desktop
- If you're installing Docker Desktop onto your Target Mac, you can use the Self Service app

Run your first container by opening the Terminal app on a Mac or PowerShell (or Git BASH) on a PC and typing into the CLI:

```
docker run -p 8000:80 --rm nginx
```

You should see output similar to this if you have never run an nginx container:

```
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
b8f262c62ec6: Pull complete
e9218e8f93b1: Pull complete
7acba7289aa3: Pull complete
Digest: sha256:aeded0f2a861747f43a01cf1018cf9efe2bdd02afd57d2b11fcc7fcadc16ccd1
Status: Downloaded newer image for nginx:latest
```

After that, you can open up a web browser and go to http://localhost:8000. You should see the default nginx welcome page.

Now you can go back to Terminal or PowerShell and hit the ctrl+c key combination to stop your docker container. If you are running docker for Windows, you will also need to issue a docker stop <id> command which we'll learn about soon.

Common docker commands

docker run

Docker run creates a container from the specified image and runs that container. In the first docker container that we ran, the docker image is the 'nginx' image. When we specify 'nginx' image, we tell docker to use that image as a starting point for our docker container.

Docker run can take many optional arguments. You can list them all by typing "`docker run --help`" into your CLI.

Let's break down the different parts of our first docker command

- docker
  - The main docker program
- run
  - The docker command
- -p 8000:80
  - The docker publish list
  - Since this particular container is a web server, we need to specify that the port being listened to by the web server within the container should be published outside the container.
  - The published port is bound to your computer's localhost interface

    - Since only one process can be bound to a port on an interface at a time, you won't be able to run multiple docker containers using the same host port (8000) but can run multiple nginx containers by publishing to different ports
    - The first number in this list is the port that docker should bind to on your host machine (your Mac or PC).
    - The second number is the port that the service within the docker container is listening on
    - You can specify multiple publishing lists if your service listens on multiple ports
- --rm
    - This flag tells docker to delete the container when it is no longer running
- nginx
    - This is the name of the image that we are using as a base to our container

docker ps

This command shows a list of all currently running docker containers along with some properties of each of those containers.

docker exec

This command allows you to run a command inside of a running docker container. Frequently, this is used to open an interactive terminal in order to run a shell inside the container. To use this command, you will need either the name of the running container or its ID; both of which can be found in the output of the "docker ps" command. This example would connect to a running container with the ID of "6a7b74abaa05" and run the "sh" shell. The "-it" part of the command states to run an interactive terminal so that we can continue to interact with the running process.

```
docker exec -it 6a7b74abaa05 /bin/sh
```

docker stop

This command will cause a running container to stop running. It is mostly used when we start our docker containers with the "-d" flag to run them in the background or when running Docker Desktop for Windows. The stop command also needs either the docker containers name or ID to reference the container that should be stopped.

```
docker stop 6a7b74abaa05
```

Docker registries and image tags

So where does that nginx docker image come from? By default, docker searches its own public registry to see if a docker image has been published by that name and tag. You can search for particular docker images by visiting the registry's website at https://hub.docker.com. If you search on that website for nginx, you will find the official nginx repository description at https://hub.docker.com/_/nginx.

Underneath the description heading, you will find a list of supported tags. These tags represent different options used to create the nginx image. For nginx, you'll notice that there are different tags for not only different release versions of nginx, but also some different configurations of nginx. Some of the nginx tags show that when using that image, you will have the module installed that will allow your web server to run perl scripts.

Many organizations, including Target, have internally hosted docker registries. The docker registry at Target is docker.target.com for docker images created by team members and hub.docker.target.com for docker images that are cached from the main hub.docker.com registry.

You can specify that you'd like to use the Target registry in the docker run command by using the location when specifying the image name.

```
docker run -p 8000:80 --rm hub.docker.target.com/nginx
```

Docker Volumes

Sometimes, it may be beneficial to persist data created by or modified by a docker container. Docker volumes allow for running containers to read, write and/or update data which is not contained within the running docker container.

Mapping Directories

In some cases, you may want your docker container to interact with files on your local filesystem. For these cases, docker allows you to map both files and directories from your local filesystem to the running docker container.

Let's say that you are currently developing a website in pure HTML and CSS and would like to see the changes you make as you make them. We can do this by modifying the command we ran for our first docker container.

First, we need to create a directory on our local computer that will store our new website. In our Documents folder, create another folder called "html_website" and add a file with the following contents:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Website</title>
  </head>
  <body>
    Hello World
```

```
  </body>
</html>
```

Now we add a new option to the docker run command in order to map that new directory to the directory where nginx will be looking for its files. From https://hub.docker.com/_/nginx we learn that nginx looks for this content in `/usr/share/nginx/html` directory. You'll see that the new `-v` option is similar to the `-p` option. The first part of that value tells docker where the directory is on our local computer, the second part tells docker where the directory should be made available from within the docker container.

```
docker run -p 8000:80 -v ~/Documents/html_website:/usr/share/nginx/html --rm nginx
```

Now that this is running, you can visit http://localhost:8000 again and you should see "Hello world".

One thing that you may notice when running the docker run command again is that there was no output. Since the docker image had already been downloaded, it did not need to download again and was able to immediately start the service.

Now make a change to the index.html file and change "World" to your name. Since the directory on our computer is mapped to a directory on the webserver, the only thing we should need to do to see our changes is to refresh our web browser.

Since directories mapped in this way are reading and writing files to our local filesystem, the Docker engine is required to also map file system commands from the container OS to the host OS. For tasks where there are not a lot of file reads/write like the example above, this will not cause an issue. For other purposes, you may want to use native docker volumes instead.

Mapping docker volumes

Another reason we may want to persist data is when we are doing some local development that involves reading and/or writing to a database. For many reasons, it may not be ideal to have to connect to a remote database during development and we may want to run the database on our local machine. Since some databases write a large number of files, this is a good reason to use a native docker volume.

Native docker volumes are created with the `docker volume create` command and given a name.

```
docker volume create my-database-data
```

Mapping this volume is done the same with as mapping a local directory except that instead of referencing a directory path, we reference the volume by name. One common database is

MongoDB. From https://hub.docker.com/_/mongo we learn that mongo keeps its database data in `/data/db`. From this we can run our new docker container with:

```
docker run -p 27017:27017 -v my-database-data:/data/db --rm mongo
```

If we were to start writing data to our database and then shut down the container, our data will be persisted in the `my-database-data` native volume. We can remove our container without any problems because no data that we care about is being created in that container.

In addition to this, volumes give us another benefit. Frequently, we will work on many projects and may need to develop against many different databases that are not all configured in the same way. By using volumes, we can easily switch between different configurations and data sets simply by changing the volume in our docker run command from one volume to another.

Dockerfile

Docker hub has lots of docker images ready to be used, but many times you may need a bit more customization for your image. Docker allows you to create your own images by specifying all of the steps required to prepare a docker image in a formatted file and then building the image based on that definition.

As we previously learned from our Docker vocabulary, a docker image is “an ordered collection of root filesystem changes”. But what does that mean? Basically, you can think of a docker image as an ordered set of revisions. An example may be something similar to confluence documents or git commits. Each time you publish an update to a confluence document, a new version is created. You can look up all previous version, compare the state of the document between versions, and revert to any previous state. You also have a root version, which is the basis for all future changes.

In the same way, docker images all start with a base document. This base document is the starting point for all of the changes that you want to make to your image and specifies another docker image. Our docker image will take our development web server that we were using earlier further and create a deployable image for our website.

Let’s create our very first custom docker image by creating a directory for all of our project files and create a file called “Dockerfile” in this directory. [Sample code can be found here]

Starting from a base image

In our Dockerfile, we first choose an appropriate base image. Since our image is going to be running a web server, we will use the nginx image as our base. To specify a base image, we use the FROM directive. You may find it useful to specify a specific version of a base image when

creating your images. This will ensure that updates to a base image that require updates to configuration items do not cause issues for you.

```
FROM hub.docker.target.com/nginx:1.17
```

Adding files

The next thing that we can do is start to add to the image all of the things that our project depends on. For our example, we will be copying both a customer NGINX config file as well as the files that make up our website. In the directory we created for our project files, create a file called "nginx.conf" and add a new directory called "website". In the website directory, we will add our index.html file as well as a graphic for our website. To add files to our docker image we will use either the COPY or ADD directives.

```
COPY nginx.conf /etc/nginx/nginx.conf
COPY website /usr/share/nginx/html
```

COPY vs ADD

The COPY and ADD directives share much of their functionality, however there are some differences where you will need to choose one over the other.

- ADD
  - Can be used on files as well as URLs
  - When adding local resources that are archives (identity, gzip, bzip2 or xz files) the files will be decompressed and the contents added to the image
- COPY
  - Can include a "--from" flag to include files from previous build stages

A full list of rules that apply to ADD and COPY can be found in the documentation. One shared rule to note is how these directives handle files and folders. You'll want to read through all of those rules. In our example, the file "nginx.conf" is written to the file location "/etc/nginx/nginx.conf". Since website is a directory, the contents of "website" are written to the directory "/usr/share/nginx/html".

Paths are important. In our example, the source paths are relative paths. You can tell they are relative paths because they don't start with "/" which refers to the root of the filesystem. For source files, this path is relative to the root of the build context. No files can be added to the docker container that are not children of the root context. We'll learn how to set the root context in a couple sections. For destinations, the path will either be an absolute path (as we have used in the example above) or relative paths relative to a specified WORKDIR. Our example above and this example have the same end result. The only difference is the first example used absolute paths, while the second uses relative paths. Setting a WORKDIR may be useful if you are copying files from multiple source paths to the same directory on the image.

```
WORKDIR /
COPY nginx.conf etc/nginx/nginx.conf
COPY website usr/share/nginx/html
```

Performing actions

Sometimes when creating a docker file the base image will not have all of the necessary software to fulfill the purpose of the image. In this case, you may want to use a package manager to install additional software. Let's say we are interested in keeping track of the country for all of our visitors. In this case, we will want to install the nginx-module-geoip package and update our log format. We can install this software using the RUN directive.

```
RUN apt-get install nginx-module-geoip
```

Since we are using the nginx container with the "1.17" tag, we can read its Dockerfile by clicking on the tag link on the nginx page of the docker hub website. This image was build using "FROM debian:buster-slim" and we know that the package manager for Debian based linux systems is aptitude, so we will run an "apt-get" command to install new software.

Keeping images small

We learned from the vocabulary section that an image "is an ordered collection of root filesystem changes", but what does that actually mean for us? In our Dockerfile, each directive we've mentioned so far (COPY, ADD, RUN) will create a new layer to our Docker image. Each layer created contains only the changes made during that directive call. Some actions we take may create temporary files that we do not need in our final docker images and those files should be deleted. Installing packages with apt-get is one of these processes that saves the archive files of the installed packages. If we were to write a Docker file that had two RUN commands, one to install the package and a second to remove the archive files, our final docker image would not have actually been reduced in size as the archive files would still exist in one of our layers. We can avoid this by chaining all of the commands to install and cleanup in a single RUN directive. Below is a small example of a chained RUN directive. A much more complex example can be found in the nginx Dockerfile.

```
RUN apt-get install nginx-module-geoip && apt-get clean
```

There are many more directives beyond FROM, ADD, COPY and RUN that can be used to create a docker image. All docker build commands can be found in the Dockerfile documentation. We won't be going over all of the commands in this session, but you are encouraged to read the documentation.

Building the docker image from Dockerfile

Once we have a Dockerfile defined how we need, the next step is to build our docker image. We will use the docker build command to accomplish this step. If we are in the directory that contains the Dockerfile as well as all of the files that will be copied/added to our docker image, we can use the following command.

```
docker build . -t my-cool-docker-image
```

Let's break down that command.

- docker
  - the main docker program
- build
  - the docker command to create the docker image
- .
  - The path to the context. Using "." means that we are using the current directory
- -t my-cool-docker-image
  - The name of our docker image

The docker build command will look for a file called Dockerfile at the root of our context and use that file to build our image. If needed, we can specify a docker file outside of our context with the "-f" flag followed by a path to our docker file. There are some use cases where doing this is quite beneficial, such as a need to create multiple docker images with different tags that share many of the same files.

Every time that you issue a docker build command, docker will look to see if any of the directives would have caused a change from the last time docker build was called. If not, docker will use the cache of the previous build and re-use any layers that it can be certain would not have changed. When setting up your Dockerfile definitions, you'll want to make sure that you put items with the least likelihood to change near the top of the dockerfile and directives with frequently changing files near the bottom. Having a good order can significantly reduce build times on subsequent rebuilds.

In order for docker to know if a particular command will cause a change from the previous layer, it needs to inspect and compare every file in the context. For some projects that install project dependencies in the context (such as a node.js project) this means that on every build there are possibly thousands or more files that need to be compared to previous states. Since we would want to re-install these dependencies from within our docker images and they would never be copied into an image manually, we can tell docker to ignore any changes from these directories with a ".dockerignore" file. This file works the same as a ".gitignore" file. In this file, we can add a line for "node_modules" and now on build and rebuilds, the node_modules directory will be ignored and build times will be greatly reduced.

Docker for services

Both of our example above run services within their docker containers. The service is a long running process that frequently provides a way to interact with the service via some sort of network connection.

All of our examples so far have started a service in our CLI and attached that service to the instance of that CLI. To quit our service, we can use the ctrl+c key combination or close our CLI window if we are running on Mac or Linux. On windows, we will also need to run docker stop. There may be instances where we will want to keep our docker containers running in the background so that they continue to run even if we close our CLI window. We may also want to run multiple containers without the need to keep a CLI window open for each container. For these scenarios, we can run our containers in detached mode by passing the -d flag to the docker run command.

```
docker run -p 8000:80 --rm -d nginx
```

You'll notice two things when you do this. First, you will be shown a long string hash once the docker container has started. This has is an ID for the running docker container. Next, you'll notice that you are shown the command prompt once again. Now you can enter another command or close the window and the docker container will continue to run.

Since the docker container is no longer attached to the CLI window, how to you stop it?

Docker has another command that can list out all of the currently running docker containers. To show all of the running docker containers, you will type:

```
docker ps
```

The list you are shown includes a "CONTAINER ID" column. The values in the container ID are the first few characters of the larger ID that you were shown when you started the container with the "-d" flag. You'll also see a "NAMES" column with a randomly generated name". Either of these can be used by docker to identify the container.

In order to stop your running container, you will issue the docker stop command and specify either the id or the name of the container you wish to stop

```
docker stop <id>
```

If helpful, you can also specify a name for the docker container rather than have it be randomly generated by passing the "--name" flag to the docker run command.

```
docker run -p 8000:80 --rm -d --name my-awesome-container nginx
```

Then you can stop the container by name

```
docker stop my-awesome-container
```

Docker for tasks

Not all dockers are used for long running services. In fact, they may be ideally suited for completing individual tasks. One use case would be when you need to manipulate or generate a file and this process requires that some dependencies have been met. These dependencies may include, but not be limited to software installations and configurations.

Let's say that you have a written a small script to download and process some information. The script itself has some dependencies and configurations. If you'd like to share this script with a colleague, you would need to not only provide them with instructions on how to install of the dependencies, you may need to help them troubleshoot the installation if they are unfamiliar with the platform you are using. Rather than all of that work, why not simply share a docker image? The docker image can be configured to install and configure all of the requirements. Since the docker container runs in isolation from the system, there should be no need to troubleshoot any environment issues since the environment can be guaranteed to be consistent across systems.

Let's create a docker image that is going to be used for the simple purpose of telling us how many days are left until the beginning of Spring. Code can be found here.

Persistent Containers

In all of the examples so far, the docker run command has been passed the "--rm" tag which removes the running container as soon as it stops. Adding that flag helps to make certain that our docker environment does not run out of disk space. More on this in the Docker maintenance section.

There are cases where we may want to not remove a container immediately after it stops. Along with all of the changes that we made while running a particular container, the containers remember all of the flags that were used to start them.

Let's take an example of the mongodb container. Since we know that we can use mongo to switch between different database setup on our local development machine, we may want to create multiple containers specific to a deployed environment. Looking at the documentation for the mongo docker image at https://hub.docker.com/_/mongo we see that there are a few different environment variables that can be passed to the docker container to control the configuration of the mongodb server. The docker command below has been put onto multiple lines for easier reading.

You might notice that this command is fairly long and depending on what additional paramaters we type, it might get longer. Having to type this out every time we want to start our

configurated mongo database may become annoying. By removing the "--rm" flag and adding the "--name" flag, we will be able to stop and start this configured container by name.

```
docker run \
-p 27017:27017 \
--name my_mongo_db_one \
-v my-database-volume:/data/db \
-e MONGO_INITDB_ROOT_USERNAME=myAdminUser \
-e MONGO_INITDB_ROOT_PASSWORD=myAdminPassword \
-e MONGO_INITDB_DATABASE=my_db \
-d \
mongo
```

When we're done working with our container for the day, we can stop the container without deleting it by using the docker stop command and referencing our container by name.

```
docker stop my_mongo_db_one
```

Then, whenever we are ready to start using this docker container again, we can start it up by name without having to remember all of the other configuration items.

```
docker start my_mongo_db_one
```

Docker Compose

Many projects that developers work on today are broken out into multiple pieces. For a given web application you may require an API that relies on a database. More advanced APIs may also rely on in-memory databases such as Redis, messaging services such as Kafka or proxy services such as GoProxy. Being able to fully replicate a deployed environment on a local machince may have many benefits: not relying on a shared development environment means that the bugs that appear during development won't affect other users since you're not using a shared environment, you are able to easily generate sample data sets to separate any concern of developers having access to production data (a common requirement when working with customer or other sensitive data), no need to worry about network outages or wifi interruptions as everything is local to your machine.

While you could go through and start up these services one at a time, you may not only have a lot of services, but you may have a lot of combination of services that you want for any particular development or testing.

Docker Compose helps save us a lot of typing by allowing us to configure an entire environment via a configuration file. This entire environment can easily be started and stopped to make certain that any and all requirements of one part of the environment are met.

To get started, we create a new yaml file and give it a name. In our example, the file we will be editing is called “docker-compose.yml”.

The first part of the docker compose file is a version number corresponding to the version of docker compose file format. The latest version is the 3.x series version and is recommended. New functionality is added with each new version so you do need to make certain that the version of the docker engine you are running is compatible with the version of the docker-compose file. A compatibility matrix is available on the docker website. Since we have installed the latest version of Docker Desktop, we will use the latest version of the docker compose file format which at the time of this writing is 3.7.

The only other required part of a docker compose file is a list of all of the services you will be managing with your docker compose file.

Let’s create a docker-compose file that starts up a mongo database as well as a web UI to manage the database so we don’t need to install any other software. The docker compose file linked above defines each of the services, their configuration, their published ports and any networks and volumes that the services use allowing the environment to be completely isolated from any other environment on our computer.

Common docker-compose commands

docker-compose up

Tells docker compose to start the environment. If no file is specified, docker compose will look for a file called docker-compose.yml in the directory where the command is run. Just like with the docker run command, you can specify a different file with the “-f” flag and start the environment as a background process with the “-d” flag.

docker-compose stop

Tells docker compose to stop the specified environment. When the environment is stopped, the containers are persisted and re-used the next time the environment is started. This is the same behavior as the persistent containers described above.

docker-compose down

Tells docker compose to stop the specified environment and remove all of the containers upon a successful stop.

Docker maintenance

By default, docker desktop will allocate a maximum of 64GB of space to store images, containers and volumes. After using docker for a while, you may notice that you are getting errors regarding not having enough space or being unable to write new files.

You can get some system wide disk usage statistics by running “docker system df”. Sample output may look something like the following:

TYPE	TOTAL	ACTIVE	SIZE	RECLAIMABLE
Images	78	47	4.416GB	3.675GB (83%)
Containers	62	1	55.63MB	45.42MB (81%)
Local Volumes	35	12	3.344GB	1.747GB (52%)
Build Cache	0	0	0B	0B

There are a few things of note in this output.

Images:
While it may seem a bit confusing from a terminology perspective, the number of images referenced in the output is a count of all of the layers that make up the images we have used so far. You will see a discrepancy between the total and active images. Docker does not have any sort of garbage collection to clean-up unused data. This shows that any images that will have been building by name will not delete outdated image layers when new images are built. The sample output above shows that on the system that generated this output, there is 3.67GB of “dangling” images (images with no parent). You can free up some disk usage by cleaning out these dangling images. Docker has the `docker rmi` command that will allow us to delete an image by specifying the image id. We can list all IDs with the `docker image ls`. This output shows us all docker images and layers, however, so we want to filter just the dangling images. We can do that with `docker image ls -f "dangling=true"`. Of course we're not going to want to copy and paste every id to delete it, so we can use a little bash magic to do it all for us with `docker rmi $(docker images -f "dangling=true" -q)`. Docker provides a prune convenience command which removes all dangling images with default options. `docker image prune`.

Containers:
Earlier we learned that containers are runtime instances of docker images and that images are immutable. The result is that every time we run a container, unless we explicitly state that the container should be removed on termination via the “--rm” flag, the created container will persist in docker storage. Over time, docker containers can begin to take a lot of space, especially if they write any sort of log files to the container file system. Unlike the docker images command, the docker container command only shows currently running docker containers by default. In order to show all containers stored in the docker storage, we need to apply the “-a” flag. When we need to clean up our containers we can delete them one at a time with the `docker container rm <id>` command or we can remove all stopped containers with `docker container rm $(docker container ls --filter "status=exited" -aq)`

Volumes:
Volumes can be cleaned much the same was as docker containers. We can list all docker volumes with `docker volume ls`. Docker volumes can be deleted with `docker volume rm <id>`. If a docker volume is current associated with any existing container we will not be able to delete it. The attempt will list the id of the associated docker container. You will first need to stop and then remove the docker container before removing the volume.

Resources and continued reading

This session covered the basics of using Docker. There is much more functionality and many more options for using docker. Here is a list of resources for continued learning, some of the resources are specific to Target, such as deploying to TAP.

- Official docker getting started guide
  - https://docs.docker.com/engine/docker-overview/
- Creating docker repositories in Artifactory
  - https://pages.git.target.com/artifactory/doc-site/02-selfservice/slack/
- Using Artifactory docker registries
  - https://pages.git.target.com/artifactory/doc-site/03-advanced/docker/
- Drone docker plugin or automated builds
  - https://pages.git.target.com/drone/doc-site/05-plugins/docker/
- Deploying an image to Kubernetes
  - https://pages.git.target.com/drone/doc-site/05-plugins/kubernetes/
- Deploying an image to TAP
  - https://pages.git.target.com/drone/doc-site/05-plugins/tap-pipline/
- Dockerfile best practices
  - https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
- Multi-Stage Docker builds
  - https://docs.docker.com/develop/develop-images/multistage-build/