Docker Presentation

What is Docker
Before Docker: install all dependencies on new machine on your
After Docker: easily to deploy same application on new VM
- Docker also offers security benefits, which I will not be covering today and Brian will be later in the semester

Docker Architecture
Docker is a client server architecture:
1. Docker CLI (command line interface) is the client
2. Docker daemon is the server

The Docker daemon runs on your host OS and does all of the heavy lifting with creating and manning containers, images, networks and volumes
The daemon has a REST API that users can use to communicate with the daemon. The Docker CLI makes the requisite REST API calls to interact with the daemon.

Docker Architecture (2)
Here's another photo illustrating the different components of Docker. Here you can see a 3rd component: the registry
Docker registry is a cloud repository that stores different Docker images that users can use in their own project. We will be talking more about how these public images are being used in Docker projects later

What is a Dockerfile?
Here we can see an example of a Dockerfile. You all received a Dockerfile for HW1. Now a Dockerfile contains the commands needed to assemble a Docker image. So the commands in the Dockerfile are ran by Docker to build a Docker image that's needed for your application

What is a Docker Image?
The obvious next question is what is a Docker image? (Read Slide) An image is an innert, immutable file that represents a snapshot of a container. It a static specification of what the container should be in runtime, including the application code inside the container and runtime configuration settings. A Docker image is the piece of software that **holds all of the dependencies and libraries necessary** to run an application. You might be thinking what is a container vs an image, but I will be covering that in a few slides.

Docker Image Layers
(Read slide)
As we can see, 1 layer is built for each instruction in the Dockerfile. Layers of a Docker image are essentially just files generated from running some command. Each instruction in a Dockerfile creates a **filesystem layer** that describes the differences in the filesystem before and after execution of the corresponding instruction. Docker is built to make an image out of

separate layers and to stack them together to form an image. Every time a layer is built, Docker stores it on the host OS. This layering system and storing of different layers is really useful in saving disk space, and reduce the time to build images as we'll see in the next few slides.

Docker Image Layers (2/3)
If we look in this example we can see a Dockerfile and the corresponding history when building that image. We can see each layer and its ID. If we go to the next slide, we have another Dockerfile that is almost the same but the last line is different. Now after building and looking at the history for this image, we will notice that the 5 out the 6 layers in this new image are the same. Only the layer corresponding to the last instruction is different. This means that previously downloaded or built layers were saved by Docker and re-used. By doing so, Docker saves disk space, and quickens the build process.

DockerHub Images
DockerHub is a repository of images that developers can use. These are stable and secure images available to the public. Instead of building our own custom Docker image, we can just pull a whole image from DockerHub. These images are based off Dockerfiles built by someone else who had already gone through the effort of figuring out what dependencies are needed to build the image of interest. Examples of common images are proxy server images, database images, etc. Check DockerHub to see what's available! It's always recommended to look for images on DockerHub first before making your own custom image, since sometimes they are stable and secure.

What is a Docker Container?
(Read slide)
A container is a runtime object. It is based on the image and represents the portable encapsulation of an environment. So a container is built off an image using the **run** command. A container is different from an image in that it is **dynamic** and can actually be written to

Docker Images vs Containers
The main difference between a container and an image is the **top writable layer**. This top writable layer allows users to interact with a container and for data to actually be written and stored in a container.
The fact that containers each have their own writable layer also multiple containers to be based off the same image. So you can have a Docker setup where you want multiple containers to built from the same image. That is doable since they can be based off the same image, but each container has its own unique **data state**. They do not share data, unless you explicitly set up your containers in a way to do.

What is a Data Volume?
Docker data volumes are specially designed directories in a container. The 2 main purposes they are used are for 1. Persistent data storage and for 2. Mounting host directories.

For data storage, there could be scenarios where we have to kill all of our containers on our machine but we don't want certain data to be lost. Data volumes are saved on a host OS even when a container is stopped. A common use case is setting up a data volume for your database container. Maybe you want to run your application on a new host but don't want to lose your data. There are ways to transfer the data from your data volume on one host OS to another, and restore the data in your database container

Mounting host directories is a very powerful use case. Usually your container is isolated from the host OS. This means you have to manually go into the container through the Docker CLI to make changes.When we mount a host directory as a data volume, what happens is if we make changes to that host directory, those same changes will be reflected somewhere in the container. This is useful in use cases with development where you regularly modify a file outside of your container on the host OS and want those changes to be reflected in the container. I will go over how that's useful later in the presentation.

Building your Own Docker Image
So we now have all of the tools to better understand what's going on in a custom Dockerfile, specifically the one handed out to all students for HW1. So here we will just go line by line explaining what's happening

1. Pulling python 3 image from Docker Hub: Django is Python based so we need an environment that has Python installed —> we elect to use an image from Docker Hub since it's stable and it's already been made for us
2. This just ensures that any logs created by the container are piped to the Terminal and printed to standard output
3. Creates a /code directory within the container
4. Sets your working directory as /code. This command sets a specific path in one spot and any RUN or COPY instructions will execute in the context of the WORKDIR.
5. Add req.txt to new directory
6. Here we didn't have to specify a path for requirements.txt, it assumes to look in /code since that's our WORKDIR. This line installs all dependencies on that text file. Note that you should use the "pip freeze" command with a virtual environment and pipe the output to a requirements.txt file to build your list of dependencies
7. This line just adds all of the project files in your current working directory to the code directory. So this container will have all of the project files for our project inside it's /code directory.

What is Docker Compose?
(Read Slide)
So a lot of applications require the use of multiple containers. For example, you could have a separate database container from a container running your web application. Before Docker Compose and later versions of Docker, developers had to explicitly create links between different containers using the Docker CLI. Since Docker Compose was made, building multi-

container Docker applications and making sure they can communicate has become much easier.

<u>Docker Container Networking</u>
If we have multiple Docker containers they have to communicate in some way. Docker's developers came up with a way to make that easy! (Depending on our use case).

The Bridge network (the default network) is installed on the OS when Docker is installed. Unless explicitly specified, each container that runs on a host OS will be connected to this network. This network is primarily used to allow containers on the same host OS to communicate.

To allow containers to communicate to one another, each container thinks it has its own set of ports. We can specify on what port each container should be listening to in order to receive information from a different container. We can also bind a container port to an actual port on the host OS to allow users in the outside world to communicate with a container. We'll see both in the next slide.

<u>Using Docker Compose</u>
Here you can see the .YML file given to everyone for HW1. The way you define a .yml file is to first specify the version, in this case we are using 2 since it's stable and we don't need any of the functionalities in the latter releases. After doing so you need to specify the services you are building. A service is a **distinct group of containers of the same image:tag**. I say group because technically, we can configure multiple containers to run for each type of service. When you specify a distinct service in a .YML file, that means a container will be spawned for that service when you run **sudo docker-compose up**.

Db: This service handles our postgres database. This is our database container

Web: This is our container running our actual Django application.
<u>Build:</u> This is used to specify where our Dockerfile is to build the image to use to spawn this container. We need to specify this since we are not basing this container off a DockerHub image.
User: This specifies what user will be running this container. Brian will talk more about this, but unless specified otherwise, Docker containers are run from a user with root privileges. From a security standpoint, you do not want this. If someone hacks your application and takes control of your container, they now have control of a user with root privileges who can do virtually anything on your container. By specifying **nobody** as our user, we are running this container as a user with non-root privileges.
command: This is used to specify what command to run when the container begins to run. So every time this container is built, or spun up, it will make migrations to the database, migrate them and run the test Django server.
Volumes: This command specifies what data volumes we would like to have in this container. Here we are mounting the folder "web-app" as a data volume and mapping it to the code directory in this container. We created a /code directory for the image for this container if you

remember our Dockerfile. What this does is, after you built your Docker containers and you want to keep developing, you can spin down your containers. Then, you can make changes to the project files in the web-app folder which containers your project files. After doing so, you can spin up your container and those changes will be reflected in your container because /web-app maps to the /code directory. If we did not do this, every time you made a change the web-app project files, you wouldn't see a change reflected in the container. This allows you to not have to continually destroy your container and rebuild it every time you want to make changes to the container during development.
Expose: This command is used to specify what port this container should listen on for connections. Here we are specifying that the web container should listen on port 8000 for connections.
Depends on: This line specifies that the database container should be spun up before the web container since web depends on it

Nginx: This service is used to build reverse proxy server container. You'll learn more about these later when you do HW2 but proxy servers essentially load balance and can cache commonly used requests. I encourage you to look more into what the advantages are of a reverse proxy server, but they are used to speed up connections to web applications among other things.

Here we can see we are pulling the nginx latest image from DockerHub to use to build this container.
Ports: Here we are binding a container port to a host port. We are binding port 8000 on the container to the host's actual port 8000. So when users connect to port 8000 on this VM, they are actually connecting to the nginx container initially. This container then directs traffic to the web service who is listening for connections on its own port 8000. If you are asking how this occurs, this is due to the configuration file that we gave you in the nginx directory.

You can see we mounted the config directory to the config file stored in the nginx container. This allows any changes we make to that file to make changes to the config file in the nginx container. Finally we made it depend on the web container, since traffic cannot be rerouted to the Django application unless that container is spun up.

Docker Compose (2.0)
This is the 2nd compose file released by Brian that solves some Docker issues that may come up. We can see that a data volume was created for the database. This is just a good practice to backup your database data. Anything written to that directory will be stored in a data volume. Docker native stores postgresql data to the folder /var/lib/postgresql/data so this setup will also save any data stored there to a data volume which can then be used to restore our database.

The main difference we see here is the fact that we have an extra service web-init.

The web-init service is similar to the **web** service but is used as a hack to make sure **docker-compose up** runs cleanly. In our old setup, if we ran **docker-compose up**, and no images were built or downloaded previously, that command would attempt to download or build each image. After building each image, Docker would attempt to spin up each container. The issue at hand is the **web** container usually is spun up before the **db** container and will attempt to connect to the **db** container and may throw a **Connection Refused** error. One way of getting around this error is by using a separate service **web-init**.

Web-init: This service is built off our custom Dockerfile but will run the initserver.sh Bash script when its container is spun up. If we look at that script, it makes migrations, migrates then and also has a while loop containing a sleep command. This sleep command gives the database container enough time to be spun up. We created this service to handle migrations as well as wait enough time for the database container to be spun up.

We also notice that we mounted the /web-app folder as a data volume to the /code directory in this service. If we look at the web container we did the same data volume mounting. This ensures that any code changes will update both of these services. Additionally, I believe any migrations that will be created in the web-init command will now be reflected in the /web-app folder which will then be reflected in the /code directory of the **web** container.

The other change we see is that the **web** container runs the **runserver.sh** Bash script. If we look at it, we see that this server just runs the run server command to actually run the Django application.

<u>Useful References</u>
Any questions?