

Why Containerization/Dockerization of Applications is Essential

1. Background and Introduction

Before Containerization, most companies and tech enthusiasts alike used and still use Virtual Machines. Software like Oracle VM VirtualBox would allow the user to create an application or a database by downloading an OS image, it can be different flavours of Linux (Ubuntu, CentOS etc.), downloading relevant application-related files, it can be an Oracle database software for database creation or framework-related files and spin up a “virtual computer” using the downloaded OS image and relevant application files via Oracle VM VirtualBox. CPU and other hardware is shared with the computer the VM is running on. This process made software development and infrastructure management more hassle-free, allowing users to speed up the development process without needing to set up an entire infrastructure. I remember using VMs when I worked in India for 1 year. Since our team was in multiple parts of India and our clients were both in India and UK, we were very given access to VMs for our work. While the process was smooth sailing for the most part, OS failures and lag in VirtualBox response time made the experience unpleasant for users, especially when the project or application had to scale up. And downloading the OS image along with software was sub-optimal for storage because an Ubuntu Linux image was 10 GB or above in size and software like Oracle database 12c or 19c was roughly 20 GB. Installing a simple OS with database software would cost 30 GB in storage. A more optimal solution was needed for quick software deployment beyond the VMs and the standard process of local development and git. The solution to this problem came with Containerization by using Docker or Kubernetes. It gave users the ability to quickly spin up a packaged application using a Docker file or Docker Compose. I would be using Docker Compose and Docker files for data storage for my AWS project.

2. Problem Definition and Significance/Importance

Given the rapidly changing nature of Application and Software Development, it has become increasingly important to have flexible software infrastructure that can function even during times of failure. “High Availability” is highly desired in the Technology world today. Containerization or Dockerization does exactly that. Dockerization would involve combining all the necessary functionality for your application into a Docker container, and quickly spinning up the application. I have used Docker for creating the data storage infrastructure for the AWS 1 project.

The Significance of Containerization in my view can be divided into

1. Size

The overall size of a docker image is only a few MB, and even after adding additional software layers, it would not for the most part cross 1 Gb, the limit for a docker container, if needed is 10 GB. Compared to that a VM with Linux OS and some software like oracle 12c would be at least 30 GB. We can add more layers to a docker container while consuming less space when compared to a VM.

2. Availability

If the operating system or the environment the docker container is running on has been comprised or is inaccessible, the container can simply be connected to another network using add-network, and this high availability means your application is protected against system failures. Connecting a container built on Linux to another Linux system is easy but connecting a Linux to windows can only happen via the windows subsystem for Linux.

In the process of writing this term paper, I have tested the docker's size and technical advantages on my IWS 1 project and will be showcasing the results in the solution section.

3. Technical Approach and Implementation

As stated in my term paper proposal, my aim was to test docker using docker-compose and docker swarm to display the flexibility and high availability and provide support for using docker for application deployment over traditional systems.

I have decided to split the technical approach and solution part into 2 sub-parts, based on the aim stated in the proposal

1. Docker Compose
2. Docker Swarm

The process for setting up and using a dockerized database container for data storage for our IWS 1 project is listed below

Docker Compose

I used Docker-Compose instead of the traditional database setup process (oracle DB manager etc.) for data storage. Docker Compose allowed me to set up a PostgreSQL database without

having to use pg. admin and set up network settings to connect to the database. And this is exactly the reason for choosing containers for the project's this allows me to plug out the database and connect it to a different network, if need be, without data loss. I set up the docker database by following this process

1. Pulled PostgreSQL DB image from the docker hub website using the “docker pull Postgres” command
2. Wrote the docker-compose file that contains the configuration to create the PostgreSQL database, the file has to be written in.YAML format, and should contain the following
 - a. Image, which is Postgres
 - b. Ports
 - c. Database username, password, and database name.,
 - d. Volume - Mount point for the file system, which can be anything the user chooses
 - e. Whether to include a database administrator tool, called Adminer to manage and query the DB.
 - f. Port for the adminer tool
 - g. The docker-compose. YAML file I wrote can be found at <https://github.com/NagarjunaKocharla/5130f2022/blob/main/Kocharla-Nagarjuna-iws1-project-progress/backend/docker-compose.yaml>
3. Since data must be persistent and must be available to be displayed, I created a volume in the docker-compose file, having volumes makes the data persistent after DB is shut down/unplugged or restarted.
4. Run the “docker-compose up -d” command to build and start the database container.
5. Run docker PS and docker inspect commands to check if the database is up and running and the mount point is active

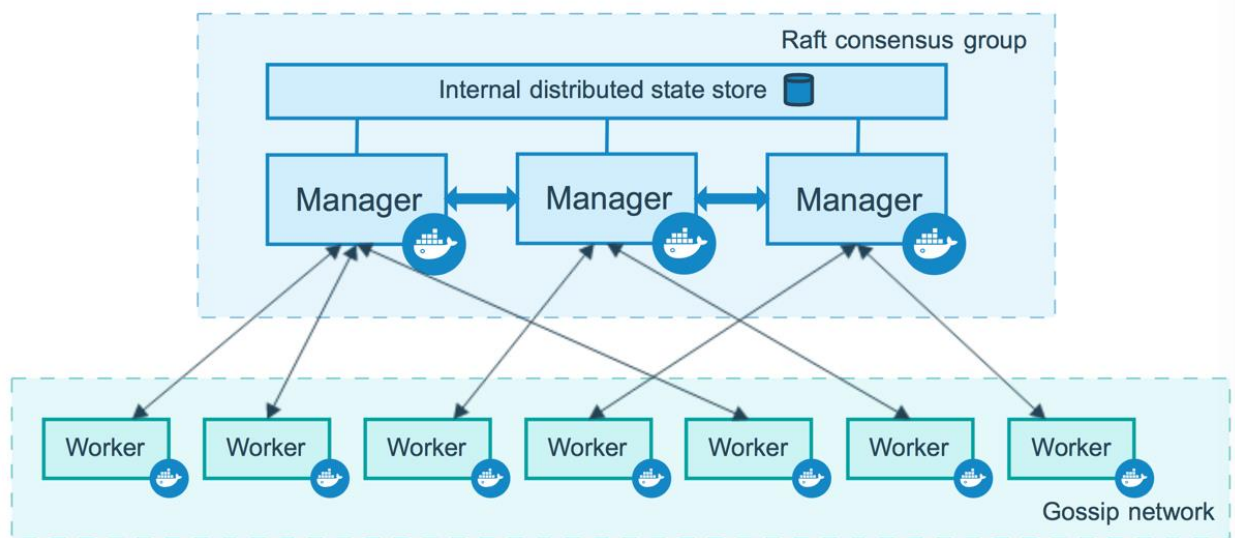
After the above steps, as you can see below the database container and DB administration tool are up and running.

```
PS C:\Users\arjun\OneDrive\Documents\GitHub\DCrypt-A-CryptoCurrency-Tracking-and-Visualization-Applicatio> docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
b2b2353f8fdd	adminer	"entrypoint.sh docke..."	4 weeks ago	Up 23 hours	0.0.0.0:8080->8080/tcp	dcrypt-adminer-1
ca10789777b8	postgres	"docker-entrypoint.s..."	4 weeks ago	Up 23 hours	0.0.0.0:5432->5432/tcp	dcrypt-database-1

Docker Swarm

High Availability of an application is highly desired in any software development cycle, but this comes with various complexities, for instance, AWS ECS provided containerized high availability where you can have multiple secure containers to fail over to if the current working container is unavailable, but AWS ECS is not open sourced and requires extensive setup and authorization, which is good in terms of security but not ideal for small scale or personal application development, and it also charges the user if a certain limit in terms of data size breaches. Docker Swarm on the other hand is open-sourced and can be set up locally. Below is a structure of the docker swarm setup



Picture downloaded from open source [official docker hub](https://docs.docker.com/engine/swarm/overview/)

Docker Swarm setup

As seen above, a docker swarm is set up by having a manager node or one or more managers who are responsible for maintaining the swarm/cluster state and making sure the internal function of the cluster is consistent across the swarm, and multiple worker nodes each of which is responsible for running a container. For instance, if you have a containerized node.js application, and data storage in a PostgreSQL container database (using the process described above under docker-compose). We can create a manager node to maintain the state across the cluster and connect these two containers (node.js application and PostgreSQL container) as worker nodes. This containerized deployment of software would provide high availability where each function that the application has is separated and brought together using a manager.

Setting up a docker swarm for testing and development is straightforward and I have done it using the below steps

1. I choose node 1 with ip 192-168-0-104 as the manager node using the command “**sudo docker swarm init --advertise-addr 172.17.255.255**”.
2. The above command would setup the swarm by making 192.168.9.104 the manager.
3. For my setup, I only have one manager as I was only testing it, but, practically, it's better to have two or more managers.
4. I went ahead and added a worker node to the swarm by running **sudo docker Swarm join --token SWMTKN-1- xxxxxx**, the xxxxxx part must be replaced by the token which can be obtained using command “**sudo docker swarm join-token worker**” (must run this on manager node).
5. The worker node is now added to the swarm and is ready to work on the tasks assigned by manager.
6. Now comes the main part where we deploy/create software that the workers have to work on, I ran **sudo docker service create --name appjs node ping docker.com** to create a node.js application, as stated in my project proposal.
7. Now the node js app is created, the task of handling the application is assigned by the manager to the worker nodes, any software like node.js, PostgreSQL database created is spread across all the worker nodes using load balancing by the manager node.
8. Below, is the manager node, where manager status is leader and the worker nodes listed. And the NodeJS application (appjs) created as a service being run on the worker nodes

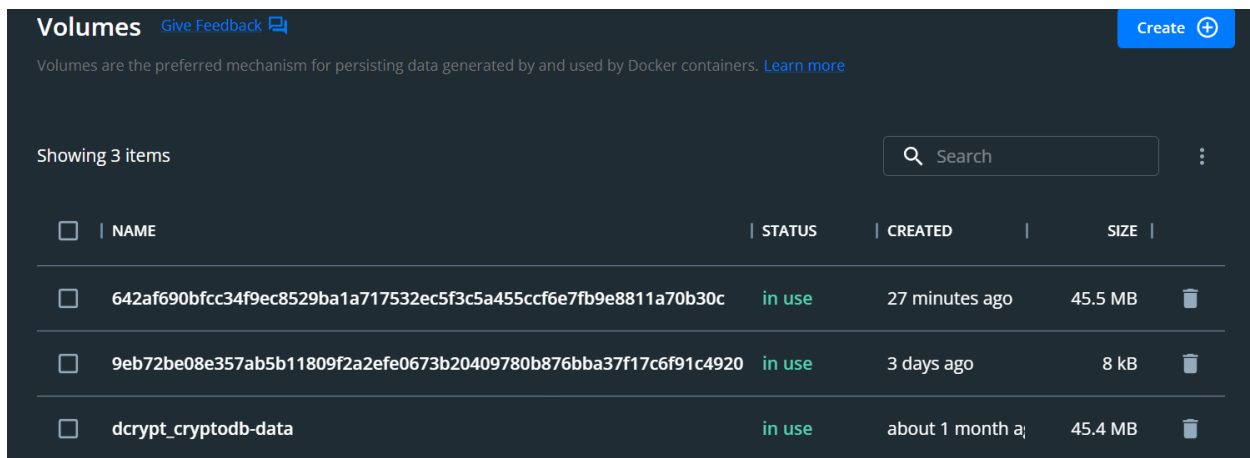
```
user@192-168-0-104:~$ sudo docker node ls
ID                HOSTNAME          STATUS    AVAILABILITY    MANAGER STATUS    ENGINE VERSION
buydvhbby9thu836ezx2pmgsp * 192-168-0-104    Ready     Active           Leader             20.10.7
92guugvnz5zggky4y9t85er0a    192-168-0-107    Ready     Active           -                  20.10.7
tdsbzyca93wyo2irtn9hl9do4    192-168-0-107    Down      Active           -                  20.10.7
t8sizvq5dhodggb5f7w0hasj4    192-168-0-140    Ready     Active           -                  20.10.21
xf6lmim62btlvscfsoq7c7kw     192-168-0-164    Ready     Active           -                  20.10.21
user@192-168-0-104:~$ sudo docker service ls
ID                NAME      MODE     REPLICAS    IMAGE           PORTS
x9tuywpgazxy     appjs     replicated 0/1          node:latest
971bt1ewl713     jsapp     replicated 0/1          node:latest
```

4. Performance and Results

Since the implementation of docker and docker swarm was done based on the **size and availability** of docker. It makes sense to explain results and benefits of utilizing docker and containers for application deployment and data storage.

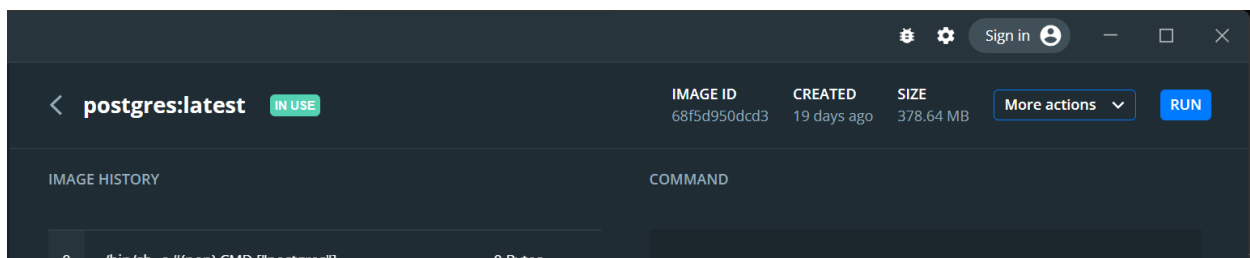
Size

Compared to creating and designing a database using a software like oracle, or MySQL where the user is expected to have a network setup, software installed, and the whole process would cost about 30 GB in size, using docker, the same functionality can be achieved with far less storage in the ranges of MBs. The main advantage comes from the fact that the setup for creating a database using docker can be done using an image of the database and its functionality, which is available to download on websites like docker hub, and only cost 10-20 mb in size. As you can see below, the docker containerized database I created and am using for my IWS1 project is available and running, using only a few mb in disk space.



The screenshot shows the Docker Volumes management interface. It features a header with 'Volumes' and a 'Create' button. Below the header, there's a search bar and a table listing three volumes. Each volume entry includes a checkbox, a long alphanumeric name, a status of 'in use', a creation time, and a size. The volumes are: 1) '642af690bfcc34f9ec8529ba1a717532ec5f3c5a455ccf6e7fb9e8811a70b30c' (45.5 MB, 27 minutes ago), 2) '9eb72be08e357ab5b11809f2a2efe0673b20409780b876bba37f17c6f91c4920' (8 kB, 3 days ago), and 3) 'dcrypt_cryptodb-data' (45.4 MB, about 1 month ago).

	NAME	STATUS	CREATED	SIZE
<input type="checkbox"/>	642af690bfcc34f9ec8529ba1a717532ec5f3c5a455ccf6e7fb9e8811a70b30c	in use	27 minutes ago	45.5 MB
<input type="checkbox"/>	9eb72be08e357ab5b11809f2a2efe0673b20409780b876bba37f17c6f91c4920	in use	3 days ago	8 kB
<input type="checkbox"/>	dcrypt_cryptodb-data	in use	about 1 month ago	45.4 MB



The screenshot shows the details of a Docker container named 'postgres:latest'. It is marked as 'IN USE'. The container's image ID is '68f5d950dcd3', it was created '19 days ago', and its size is '378.64 MB'. There are buttons for 'More actions' and 'RUN'. Below this, there's a section for 'IMAGE HISTORY' and 'COMMAND', showing the command used to start the container: '/bin/sh -c #(nop) CMD ["postgres"]'.

NAME	IMAGE ID	CREATED	SIZE
postgres:latest	68f5d950dcd3	19 days ago	378.64 MB

The database container setup, using docker compose was about 45.4 mb in size as seen above and the database has 378 MB worth data, which is negligible compared to the data storage and setup requirements, traditional database setup would need.

Since a limit to the number of transactions cannot be placed when building the applications, it made sense for my project to save space when possible, during the database build and creation, as having a database with large storage capacity is unfeasible and only other alternative like AWS is paid.

High Availability

Failover is essential in the modern software development cycle, and high availability of worker ready to handle the application when there is data compromise or failure. Using Docker Swarm, I created a cluster of nodes to handle a NodeJS application, as seen below

```
user@192-168-0-104:~$ sudo docker node ls
ID                                HOSTNAME      STATUS    AVAILABILITY  MANAGER STATUS  ENGINE VERSION
buydvhbby9thu836ezx2pmgsp *      192-168-0-104 Ready     Active        Leader         20.10.7
92guugvnz5zggky4y9t85er0a       192-168-0-107 Ready     Active        -              20.10.7
tdsbzyca93wyo2irt9hl9do4        192-168-0-107 Down      Active        -              20.10.7
t8sizvq5dhodggb5f7w0hasj4       192-168-0-140 Ready     Active        -              20.10.21
xf6lmim62btlvsjcfsoq7c7kw       192-168-0-164 Ready     Active        -              20.10.21
user@192-168-0-104:~$ sudo docker service ls
ID                NAME      MODE     REPLICAS  IMAGE      PORTS
x9tuywqazxy      appjs     replicated 0/1        node:latest
971bt1ewl713     jsapp     replicated 0/1        node:latest
```

As seen, the NodeJS application I create (appjs) is managed by the manager node 192.168.0.104 and the manager using load balancing, assigned the application as a service to the 3 worker nodes in the swarm 192.168.0.107, 192.168.0.140, 192.168.0.164. The app is running on the three worker nodes, failure of any one node would not cause an outage on the application, and provides high availability

5.Limitations

Docker container and swarm are better suited for development and testing more than production. The reason/s for this are

1. Docker has very limited health checks in place for container health compared to AWS, so limited health checks is a bottleneck for production deployments
2. Replacing a container would need downtime, and cannot be done on a rolling basic, development and test can sometimes suffer from downtime, but production must available all the time
3. Due to it's easy of use and less size, the containers built using docker were designed to provide to optimize size requirements over speed and network capabilities, but there might be requirements when the later is preferred and there are other options like AWS that do better

6.Conclusion

In the process of writing this paper and working on the IWS1 project. I researched about and implemented docker/containerized solutions for database and software management instead of an inhouse database system like oracle dbms. And lent support to the benefits and importance of using docker with regards to size and high availability. In the process of supporting docker, I practically implement docker and docker swarm and displayed the results above.

While dockerization is not comprehensive and has bottlenecks, the ease of use and setup is a major advantage and removes the burden of extensive setup a service like AWS needs, for high availability and space.

7.References

1. https://repository.stcloudstate.edu/cgi/viewcontent.cgi?article=1091&context=msia_etds
2. <https://docs.docker.com/>
3. <https://www.geeksforgeeks.org/why-should-you-use-docker-7-major-reasons/>
4. <https://docs.docker.com/compose/compose-file/compose-file-v3/>