# Practice M3: Container Orchestration

For this practice we will use an infrastructure like this:

A diagram of a network

AI-generated content may be incorrect.

All exercises that follow assume that we are working in an environment with both **VirtualBox** and **Vagrant** installed

For the VM we can use either an **AlmaLinux OS 9.x** based, or **Debian 12.x** based **Vagrant** box

*Please note, that some commands end with this* ***\*** *symbol. This is because the command is long and does not fit on one line. Written this way makes it more readable. When you type in such commands, you can omit the* ***\*** *symbol and continue with the rest of the command*

## Part 1: Distributed Applications & Docker Compose

Navigate to the folder for **part 1**. This should be either **m3/almalinux/1**, or **m3/debian/1**

Start the VM

**vagrant up**

After the machine is up and running, open an SSH session to it

**vagrant ssh**

Then, we must prepare and convert our application to a containerized one

*DISCLAIMER: Normally, this is done via fork. Instead, we will simulate different situation. We will pretend that we created the application from zero and then while evolving, we want to track the changes with* ***git*** *and* ***Git Hub***

### Preparation #1

Make sure that you have an account in **Git Hub.** If you do not have, create one

Then log on and go to **Settings**

Next, go to **Developer Settings** and click on **Personal access tokens**

We can select either the **Fine-grained tokens** option which gives us more control, or the **Tokens (classic)** option

Select the second (***classic***) one

Then click on **Generate new token** drop-down and select again the ***classic*** option

If asked to authenticate, you must enter your password and confirm

Fill the form – enter a note, change the expiry date and give full **repo** permissions

Click **Generate token**

Finally, copy the generated token *(you won’t see it again, so keep it somewhere safe)*

Then, create an empty repository *(for example* ***bgapp****,* ***myapp****,* ***demoapp****, etc.)*

**Do not close the browser yet**. Next, return in our terminal session

Let’s configure our name and email as they will be part of all interactions with **Git Hub**

**git config --global user.name "<user name>"**

**git config --global user.email "<user mail>"**

We can check the configuration with

**git config -l**

### Preparation #2

Let’s clone the repository but change the folder name *(use the same name as the one of the new repository)*

**git clone https://github.com/shekeriev/do-sample-app mybgapp**

Navigate to the folder of the cloned project *(****mybgapp*** *in our example or whatever you picked up)*

**cd mybgapp**

Check the structure

**tree**

Check it again but this time with the hidden objects as well

**tree -a**

Clean up the folder by removing the git system data (we want a fresh/clean project as if we just started working on it). **This is not something we usually do**, especially if we want to collaborate with others. **This is an exception**

**rm -rf .git**

And then initialize the **README.md** file

**echo 'My BGApp Project' > README.md**

Now, we have our clean set of files as if we just coded them 😊

Next, let’s initialize our local **git** repository

**git init -b main**

Now, we should establish a link between the local repo and the remote (in **Gut Hub**) one

Copy the information provided on the repository creation page (select a protocol – **HTTPS** or **SSH**). Let’s use **HTTPS**

Execute the following command *(copy it from the new repository page that you still have open)*

**git remote add origin https://github.com/shekeriev/mybgapp.git**

Verify the remote **URL**

**git remote -v**

Now, let’s start tracking the existing files

First, we may check the status of our local repository

**git status**

Okay, we are on the **main** branch, and we have plenty of untracked files

Add all files

**git add .**

And then check the status again

**git status**

Now, they are tracked but not committed yet. Let’s do the initial commit

**git commit -m "Initial commit"**

And push the changes to **Git Hub**

**git push origin main**

Enter the username and the generated token (just paste it, it won’t appear)

We can remember the token locally for an hour with

**git config --global credential.helper "cache --timeout=3600"**

Go to the browser and check if the files are there

We can check the status and the logs on the command line as well

**git status**

**git log**

### Preparation #3

We can consult with **Docker Hub** to select the most suitable base images for our needs

After quick research, we can pick the following

* **php** (and more specifically, its **8-apache** tag)
* **mariadb** (and more specifically, its **10.7** tag)

Let’s start with the web tier

For the web part, we will try to produce an image that is as generic as possible

Create a file named **Dockerfile.web** *(yes, we will violate the naming convention a bit)* with the following content

**FROM php:8.0-apache**

**RUN docker-php-ext-install pdo\_mysql**

Save and close the file

Now, build a local image with the following command *(with an extra option due to the different name of the file)*

**docker image build -t img-web -f Dockerfile.web .**

Once the build succeeds, test the image by running container out of it

**docker container run -d --name web -v $(pwd)/web:/var/www/html -p 8080:80 img-web**

Test the component either locally with

**curl http://localhost:8080**

Or by opening a browser on the host and navigating to **http://localhost:8080**

In either case, it should be working

Stop and remove the container

**docker container rm --force web**

Add the changes

**git add Dockerfile.web**

And commit them

**git commit -m "New Dockerfile for the web component"**

Finally, we can push those changes

**git push origin main**

We are done with the web component

For the database, we will follow different path. We will inject our data in the image

Then create a file named **Dockerfile.db** with the following content

**FROM mariadb:10.7**

**ADD ./db/db\_setup.sql /docker-entrypoint-initdb.d/init.sql**

Save and close the file

Now, build a local image with the following command

**docker image build -t img-db -f Dockerfile.db .**

Once the build succeeds, test the image by running container out of it

**docker container run -d --name db -e MYSQL\_ROOT\_PASSWORD=12345 img-db**

Now, we should test if our data is there

Start a session in the container with

**docker container exec -it db bash**

Then open the **CLI**

**mysql -u root -p**

And enter the password you used to start the container. For example, **12345**

Make sure that **UTF8** is used

**set names 'utf8';**

Then retrieve the list of databases

**show databases;**

Our database is here. So, let’s switch to it

**use bulgaria;**

And check its tables

**show tables;**

Okay, the **cities** table is the only one here. Ask for its data

**select \* from cities;**

Everything is here. Close the session to the database

**quit**

And then to the container

**exit**

Stop and remove the container

**docker container rm --force db**

Add the changes

**git add Dockerfile.db**

And commit them

**git commit -m "New Dockerfile for the db component"**

Finally, we can push those changes

**git push origin main**

We have the two components of our application ready, and we can move forward

### Distributed Applications

When deploying distributed application in a containerized environment usually every component runs in its own container

So, we must ensure that the related components can communicate together

#### Linking

*This is considered a legacy option, but we will try it anyway*

One of the options to provide containers with a way to communicate is to link them

Let us first prepare the environment

Edit the **web/config.php** file and place the following there

**$database = "bulgaria";**

**$user = "web\_user";**

**$password = "Password1";**

**$host = "db";**

Save and close the file

Add the changes to **Git Hub** (note that storing users and passwords is NOT a good idea)

**git add web/config.php**

**git commit -m "Changed configuration settings"**

**git push origin main**

We are ready to continue by starting the two containers

First, let’s start the database container

**docker container run -d --name c-db -e MYSQL\_ROOT\_PASSWORD=Parolka-12345 img-db**

And then the web container

**docker container run -d --name c-web -p 8080:80 -v $(pwd)/web:/var/www/html --link c-db:db img-web**

Note the name, we used in the link. It is the same as we stated in the **config.php** file

Check the list of running containers

**docker container ls**

Now, open a browser tab on your host and navigate to the following **URL** (adjust the address to match yours):

<http://192.168.99.100:8080>

We should see a sample **PHP** page that reads data from a **MySQL/MariaDB** database

Before we stop the containers, let us enter the **c-web** one and check how the link is expressed

**docker container exec -it c-web /bin/bash**

For this, we must execute the following command

**cat /etc/hosts**

So, there is a record in the **hosts** file

Close the session

**exit**

Now, we can stop and remove both containers

**docker container rm --force c-web c-db**

#### Isolated Network

Alternatively, we can create an isolated network and, in this way, allow the containers to communicate

Create a new network

**docker network create --driver bridge app-network**

Now, start the first container

**docker container run -d --net app-network --name db -e MYSQL\_ROOT\_PASSWORD=Parolka-12345 img-db**

Note the name, we used. It is the same as we stated in the **config.php** file

And then, the second one

**docker container run -d --net app-network --name web -p 8080:80 \**

**-v $(pwd)/web:/var/www/html img-web**

Check the list of running containers

**docker container ls**

Now, return to the browser tab and refresh the page, we must see exactly the same result achieved in different way

Enter one of them and check the content of the **/etc/hosts** file again. This time, we will do it in a different way

**docker container exec -it web cat /etc/hosts**

There isn’t any record for the other host

Let us stop and remove the containers

**docker container rm --force db web**

### Docker Compose

For the cases in which we want to manage a group of containers, we can use the **Docker Compose**

We can choose between a standalone application (**docker-compose**) and an integrated (into **Docker CLI**). Both are now referred to as **Compose v2** and the main difference is how we install and interact with them

Select the one you prefer. For the tasks in this practice, it won’t make any difference except on how the commands are written

Various installation options are described here: <https://docs.docker.com/compose/install/>

**Note that the following two sections are provided for completeness. We do not need to execute any of the steps there as the Docker Compose plugin gets installed automatically (as least most of the time)**

#### Installation (standalone) (skip it, it is legacy)

More information can be read here: <https://docs.docker.com/compose/install/other/#install-compose-standalone>

To eliminate the discrepancies caused by the different environments, we will install it on the **Docker** host

So, on a 64-bit Intel-compatible Linux distribution, we must execute the following to download it

**curl -SL https://github.com/docker/compose/releases/download/v2.11.2/docker-compose-linux-x86\_64 -o /usr/local/bin/docker-compose**

*Should we want another architecture, we should change the* ***x86\_64*** *string to the appropriate one. For example,* ***aarch64****. For this, we can use the output of the* ***uname -m*** *command*

And then, make it executable

**sudo chmod +x /usr/local/bin/docker-compose**

In any way, execute it to test if it is working fine

**docker-compose version**

If due to some reason the above fails, we should check the PATH variable and perhaps create a symbolic link

**sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose**

#### Installation (integrated) (skip it)

Instructions on how to install this one can be found here: <https://docs.docker.com/compose/install/linux/>

##### From a Repository

If we already have the **Docker** repository registered, then it is a matter of a just issuing the following for **Debian**-based distributions

**sudo apt-get update**

**sudo apt-get install docker-compose-plugin**

And for **Red Hat**-based distributions this

**sudo dnf install docker-compose-plugin**

*Please note that the package may have a different name on some distributions. For example,* ***docker-cli-compose***

Finally, test if it is working

**docker compose version**

##### Manual Installation

First, we must create a sub-folder in our home folder

**mkdir -p ~/.docker/cli-plugins/**

Then download the plugin there

**curl -SL https://github.com/docker/compose/releases/download/v2.37.2/docker-compose-linux-x86\_64 -o ~/.docker/cli-plugins/docker-compose**

*Should we want another architecture, we should change the* ***x86\_64*** *string to the appropriate one. For example,* ***aarch64****. For this, we can use the output of the* ***uname -m*** *command*

And make it executable

**chmod +x ~/.docker/cli-plugins/docker-compose**

Finally, test if it is working

**docker compose version**

#### Working (Experiment #1)

Let’s start with something simple

Create a separate folder in the home folder of the **vagrant** user

**mkdir -p ~/compose/web**

And navigate to it

**cd ~/compose**

There create a simple **web/index.php** file with the following content

**<?php**

**print "<h3>Hello Docker Compose!</h3>\n";**

**print "<hr />\n";**

**print "<small><i>Served by: <b>".gethostname()."</b></i></small>\n";**

**?>**

Now, we can create a **Dockerfile** with the following content

**FROM php:8.0-apache**

**COPY ./web /var/www/html**

And finally, we can test if the image will be built

**docker image build -t test .**

Test if we can run a container

**docker container run -d --name test -p 8080:80 test**

And see if the result is as expected

**curl http://localhost:8080**

Stop and remove the container

**docker container rm --force test**

And then remove the image

**docker image rm test**

We are ready to create our first compose file

Create **docker-compose.yaml** file with the following content

**version: "3.8"**

**services:**

**web:**

**build: .**

**ports:**

**- 8080:80**

*The first line (****version: "3.8"****) could be skipped with the recent versions of Compose*

Test the file with

**docker compose config**

Or with *(if you are using the stand-alone Compose)*

**docker-compose config**

*From now on, only the newer version (without the dash) of the commands will be used*

Initiate the build of the image

**docker compose build**

And check the list of images

**docker image ls**

Start the services with

**docker compose up**

Our session will be attached to the session of the running container

Test if the app is working in a browser tab (<http://localhost:8080>)

It should work

Now, press **Ctrl+C** to stop the service

Should we want, we can start the service again but this time in detached mode with

**docker compose up -d**

And then check the list of working containers

**docker container ls**

We can use the following to get the list of services

**docker compose ps**

And this one to see the list of processes by service

**docker compose top**

Should we want to see the logs (as we saw when our session was attached), we can use

**docker compose logs**

Let’s stop the service

**docker compose down**

#### Working (Experiment #2)

Now, we can assemble our multi-container application (from part 1) into one artefact which we can manage easily

Make sure that you are still in the folder where the project was cloned *(for example,* ***mybgapp****,* ***demoapp****, etc.)*

Create a **docker-compose.yaml** file with the following content

**services:**

**web:**

**build:**

**context: .**

**dockerfile: Dockerfile.web**

**ports:**

**- 8080:80**

**volumes:**

**- "/home/vagrant/<app-folder>/web:/var/www/html:ro"**

**networks:**

**- app-network**

**depends\_on:**

**- db**

**db:**

**build:**

**context: .**

**dockerfile: Dockerfile.db**

**networks:**

**- app-network**

**environment:**

**MYSQL\_ROOT\_PASSWORD: "12345"**

**networks:**

**app-network:**

Make sure that you change the **<app-folder>** to a value that matches your situation

Note that there are many differences compared to our simple compose file we just did:

* two services – one for each component of the application
* volume to mount the php files
* defined network
* using custom **Dockerfile**

Test the validity of the file

**docker compose config**

Initiate the build of the image

**docker compose build**

Start the services in detached mode with

**docker compose up -d**

Open a browser tab and navigate to <http://localhost:8080> to test the application

Please note that at first the database may not appear, but this will change in a while

List the running compose projects (we expect to see one)

**docker compose ls**

Let’s experiment again with the commands but this time in the context of two services

List them

**docker compose ps**

Ask for their processes

**docker compose top**

Check their logs

**docker compose logs**

Now, let’s stop the database service (**db**)

**docker compose stop db**

Now, return to the browser and refresh the page. No connection to the database message should be displayed

Return to the command line and start it again

**docker compose start db**

Now, return to the browser and refresh the page. Everything should be just fine

Now, let us stop and delete both containers simultaneously

**docker compose down**

We can commit this initial version as it is working just fine

**git add docker-compose.yaml**

**git commit -m "Initial docker-compose.yaml file"**

**git push origin main**

#### Working (Experiment #3)

Now, let’s modify our compose file and separate the configuration data from the rest

Let’s assume that we want to store separately the database password and the path to the php files

Should you want, you can create a backup copy first

Open the file for editing and modify it to match the following

**services:**

**web:**

**build:**

**context: .**

**dockerfile: Dockerfile.web**

**ports:**

**- 8080:80**

**volumes:**

**- "${PROJECT\_ROOT}:/var/www/html:ro"**

**networks:**

**- app-network**

**depends\_on:**

**- db**

**db:**

**build:**

**context: .**

**dockerfile: Dockerfile.db**

**networks:**

**- app-network**

**environment:**

**MYSQL\_ROOT\_PASSWORD: "${DB\_ROOT\_PASSWORD}"**

**networks:**

**app-network:**

Next, we must create an environment (**.env**) file with the following content

**PROJECT\_ROOT=/home/vagrant/<app-folder>/web**

**DB\_ROOT\_PASSWORD=12345**

Make sure that you change the **<app-folder>** to a value that matches your situation

The rest of the steps are the same

Test the validity of the file

**docker compose config**

Initiate the build of the image

**docker compose build**

Start the services in detached mode with

**docker compose up -d**

Open a browser tab and navigate to <http://localhost:8080> to test the application

Now, let us stop and delete both containers simultaneously

**docker compose down**

As it is not a good idea to keep sensitive information (the .env file in our case) in a repository, we will make a few more adjustments

First, let’s execute the following to see what are the pending changes

**git status**

Then, add a special file to instruct git to ignore some files (in our case just the **.env** file)

**echo '.env' > .gitignore**

Ask for the status again

**git status**

Now, the **.env** file does not appear in the list

Usually, when omitting such files, we add example files to their place and include those in the repository

Create a copy of the **.env** file

**cp .env .env.example**

And adjust its content to match this

**PROJECT\_ROOT=/home/vagrant/<app-folder>/web**

**DB\_ROOT\_PASSWORD=<db-password>**

Let’s store the changes

**git add docker-compose.yaml**

**git add .env.example**

**git add .gitignore**

**git commit -m "Config data moved to .env"**

**git push origin main**

We are done (in a way) with this. Of course, there is plenty of room for improvement here

## Part 2: Swarm Cluster

Before we continue let us free some resources

Exit from the **Docker instance** if you are still in

Stop and remove the machine

**vagrant destroy --force**

#### Infrastructure Preparation

Navigate to the folder for **part 2**. This should be either **m3/almalinux/2**, or **m3/debian/2**

Explore the **Vagrantfile** file. It will create three identical machines

Should you need, adjust the memory or the count of the machines (we need at least two)

Once done with the adjustments, bring up the whole set of new **Docker** nodes

**vagrant up**

Check their status

**vagrant status**

Keep in mind that in order to be able to create the cluster, we must ensure that communication is allowed between the machines on the following ports

* **2377/tcp** for cluster management communications
* **7946/tcp** and **7946/udp** for communication among nodes
* **4789/udp** for overlay network traffic

Alternatively, the firewall should be stopped

Let’s establish a session to the first machine

**vagrant ssh docker1**

Initialize it as the first node of the cluster

**docker swarm init --advertise-addr 192.168.99.101**

A sample command (including the token) to join the worker nodes is shown

*If we lost the token, we could always ask for it with*

***docker swarm join-token -q worker***

Copy the result of the above (***docker swarm init***) command and exit the instance

**exit**

Now, enter the second instance

**vagrant ssh docker2**

Execute the copied command to join the second node as a worker

**docker swarm join \**

**--token <token-text> \**

**--advertise-addr 192.168.99.102 192.168.99.101:2377**

Then exit the session

**exit**

And finally, repeat the procedure on the third node

**vagrant ssh docker3**

Join the third node to the cluster

**docker swarm join \**

**--token <token-text> \**

**--advertise-addr 192.168.99.103 192.168.99.101:2377**

Close the session to the third node

**exit**

We can return on the first host

**vagrant ssh docker1**

And ask for the nodes list

**docker node ls**

#### Start a Service

Log on to the first machine

To start a simple service, we can execute

**docker service create --replicas 1 --name pinger alpine ping softuni.bg**

We can ask for a list of the running services

**docker service ls**

To see the details of a service, execute

**docker service inspect pinger**

There is an alternative, more beautiful version

**docker service inspect --pretty pinger**

We can see the tasks of a service and where (on which node) they are running

**docker service ps pinger**

Let us increase the number of tasks to 5

**docker service scale pinger=5**

And check where they are running

**docker service ps pinger**

#### Node Maintenance

We may need to do some maintenance on a node

First, we must drain the node and then do the maintenance. This will move all containers that are running on it to the other nodes

Draining is done in the following way

**docker node update --availability drain docker2.do1.lab**

Ask for information about the drained node

**docker node inspect --pretty docker2.do1.lab**

Note the **State** and **Availability** pair of fields

Check details about the service we have

**docker service ps pinger**

We can see that the requested number of copies of the service is still the same but redistributed

Bring back the host as part of the cluster

**docker node update --availability active docker2.do1.lab**

Ask again for detailed information for the node

**docker node inspect --pretty docker2.do1.lab**

Note the **State** and **Availability** pair of fields

Check again the service

**docker service ps pinger**

As we can see, the node is back, but the tasks were not redistributed again

We can force the redistribution of the tasks

**docker service update --force pinger**

The above command achieves the requested but with the price of **restarting the service**

There are situations in which this is **unacceptable**

Instead, we can modify the service definition when we start it

**docker service create --replicas 1 --name pinger \**

**--update-delay 10s alpine ping softuni.bg**

We can even add a parameter stating how many tasks can be updated simultaneously with **--update-parallelism**

We can stop the service

**docker service rm pinger**

#### Service with Published Port

Let’s start a mini project that we will use for this and the next two exercises

##### Preparation

Being on the first node, create a folder

**mkdir -p ~/app/web**

Enter the **app** folder

**cd ~/app**

And create **web/index.php** file with the following content

**<?php**

**print "<h3>Hello Docker Swarm!</h3>\n";**

**if (getenv('APP\_MODE')) print "Running in ".getenv('APP\_MODE')." mode.<br />\n";**

**print "<hr />\n";**

**print "<small><i>Served by: <b>".gethostname()."</b></i></small>\n";**

**?>**

Now, we can create a **Dockerfile** with the following content

**FROM php:8.0-apache**

**COPY ./web /var/www/html**

And finally, we can test if the image will be built

**docker image build -t swarm-app .**

Test if we can run a container with

**docker container run -d --name web -p 8080:80 -e APP\_MODE='test' swarm-app**

Now, check the app with

**curl http://localhost:8080**

Okay, it is working. Remove the container with

**docker container rm --force web**

Now, we must push our image to a registry accessible by all members of the cluster

One such registry could be the **Docker Hub**

Of course, we must have a registration there (it is free and can be done in seconds)

If you do not have and do not want to create one, then you can skip the next few steps and continue with the next section – ***deploy the service*** *(remember to substitute* ***<username>*** *with* ***shekeriev*** *as it has the image published)*

Assuming that we have one, let’s use it to login to the registry

**docker login**

Enter the credentials

Then tag the image against our profile in **Docker Hub**

**docker image tag swarm-app <username>/swarm-app**

Check the list of locally available images

**docker image ls**

And then push the image to the registry

**docker image push <username>/swarm-app**

##### Deploying the service

Now, we can create the service with

**docker service create --name app --publish published=8080,target=80 <username>/swarm-app**

This will result in a single task. Let’s check

**docker service ps app**

Let’s open a browser tab on the host and visit <http://192.168.99.101:8080>

The app should be there

We can scale it for example, to 4 replicas

**docker service scale app=4**

Then check the service catalog

**docker service ls**

And the tasks list of the service

**docker service ps app**

Return to the browser tab and refresh a few times

Now, stop and remove the service

**docker service rm app**

Let’s start it again, but this time distribute its tasks using a different strategy – global instead of replicated

**docker service create --name app --publish published=8080,target=80 --mode global --env APP\_MODE=global <username>/swarm-app**

Ha, it started three tasks, and we did not specify this explicitly

So, with the **global distribution mode**, the **number of tasks** **equals** the **number of nodes**

We changed something else as well. We added an environment variable to alter application’s behavior

Open a browser tab and navigate to <http://192.168.99.103:8080> *(yes, a different address, it is the third node)*

Refresh a few times. We should see only three different host names

Let’s check the tasks

**docker service ps app**

And remove the service

**docker service rm app**

#### Service with Custom Data (Filesystem Based)

Okay, environment variables are one way to pass data to the tasks in a service

There are situations in which we would like to attach a filesystem mount to the service

We can use either **data volume** or **bind mount**

In this case, we will go with the latter

Let’s prepare the playground

Being on the first node, create a folder

**mkdir -p ~/swarm**

And create a simple **index.php** file in it with the following content

**<?php**

**print "Node: <b>".gethostname()."</b>";**

**?>**

We will use it to overwrite the application’s **index.php** file

Now, distribute it to the other nodes

**scp -r ~/swarm docker2.do1.lab:.**

**scp -r ~/swarm docker3.do1.lab:.**

And finally, start the service with 5 replicas using the following command

**docker service create --name app --publish published=8080,target=80 --replicas 5 --mount type=bind,src=$(pwd)/swarm,dst=/var/www/html,readonly <username>/swarm-app**

Check the result with

**docker service ps app**

And then in the browser by navigating to <http://192.168.99.101:8080>

Refresh a few times and then remove the service

**docker service rm app**

#### Service with Custom Data (Docker Config Based)

Instead on mounting the file, we can upload it to a **Docker Config** resource

Config values can be generic strings or binary content (up to **500 kb** in size)

Keep in mind that they are not encrypted and thus you should not store sensitive data inside

Should you have to, you must use **Docker Secrets** resource

Now, to our case, create a configuration based on the **index.php** file we created earlier

**docker config create custompage swarm/index.php**

We can list the available secrets with

**docker config ls**

Of course, we can inspect them

**docker config inspect custompage**

Now, create a service that will consume this configuration

Start the service with 3 replicas using the following command

**docker service create --name app --publish published=8080,target=80 --replicas 3 --config src=custompage,target=/var/www/html/index.php <username>/swarm-app**

Check the result with

**docker service ps app**

And then in the browser by navigating to <http://192.168.99.101:8080>

Refresh a few times and then remove the service

**docker service rm app**

#### Group of Services (Stack)

Make sure that you are still logged on the master node

Clone the project that you created earlier

**git clone https://github.com/<username>/<repo>.git**

Enter the project’s folder

**cd <repo>**

Log on to **Docker Hub**

**docker login**

Build the two images. First the one for the web component

**docker image build -t <username>/bgapp-web -f Dockerfile.web .**

And then the one for the database

**docker image build -t <username>/bgapp-db -f Dockerfile.db .**

Now push them

**docker image push <username>/bgapp-web**

**docker image push <username>/bgapp-db**

Now, create a new compose file

**cp docker-compose.yaml docker-compose-swarm.yaml**

Open the new version and make sure it looks like this

**services:**

**web:**

**image: <username>/bgapp-web**

**deploy:**

**replicas: 5**

**ports:**

**- 8080:80**

**volumes:**

**- "/home/vagrant/<app-folder>/web:/var/www/html:ro"**

**networks:**

**- app-network**

**db:**

**image: <username>/bgapp-db**

**networks:**

**- app-network**

**environment:**

**MYSQL\_ROOT\_PASSWORD\_FILE: /run/secrets/db\_root\_password**

**secrets:**

**- db\_root\_password**

**secrets:**

**db\_root\_password:**

**external: true**

**networks:**

**app-network:**

The most significant difference is that we substituted the build instructions with image instructions and added secrets

Wait. What are those secrets? Where they came from? Let’s explore a bit

Go to the home folder of the user

**cd ~**

Execute the following to create a secret

**echo 'Top Secret' | docker secret create secret1 -**

Then create a file named secret2.txt with the following command

**echo 'File-based secret' > secret2.txt**

And create a secret out of it

**docker secret create secret2 secret2.txt**

Now, let’s list those secrets

**docker secret ls**

And explore one of them

**docker secret inspect secret1**

Not much to see here

Let’s start a container and see how the secret is seen inside it

First, create and run the container (we are in swarm mode, so it is a little bit different now)

**docker service create --name secret-test --secret secret1 --secret secret2 alpine sleep 1d**

Then, list the content of a special folder inside the container

**docker container exec $(docker container ls --filter name=secret-test -q) \**

**ls -l /run/secrets**

And why not, see the content of one of the files

**docker container exec $(docker container ls --filter name=secret-test -q) \**

**cat /run/secrets/secret1**

Let’s stop the container (the service in our case)

**docker service rm secret-test**

And remove the secrets

**docker secret rm secret1 secret2**

Now, we are ready to create the secret that we referred to in the Swarm Compose file

**echo '12345' | docker secret create db\_root\_password -**

Okay, we may return to our code (don’t forget to switch to the folder) and commit the changes to the project

First, we may need to repeat the get-related configuration steps like setting user, email, etc.

Once done, we can check the status and see what we have pending (it should be just the new compose file)

**git status**

And then confirm and publish the changes

**git add docker-compose-swarm.yaml**

**git commit -m "Added compose version for swarm"**

**git push origin main**

Now, clone the project on the other two nodes by executing

**ssh docker2 -- git clone https://github.com/<username>/<repo>.git**

**ssh docker3 -- git clone https://github.com/<username>/<repo>.git**

Of course, you can log on each node and execute the cloning command locally

Do not forget to change both **<username>** and **<repo>** to match yours

Once done, make sure that you are back on the first node

Then, make sure that you are in the project’s folder and execute the following to start the stack

**docker stack deploy -c docker-compose-swarm.yaml bgapp**

Once the stack is deployed, we can list available stacks

**docker stack ls**

And then list the services in the **bgapp** stack

**docker stack services bgapp**

Check information about the stack

**docker stack ps bgapp**

Perhaps, we should wait for the services to start

We can open a browser tab on the host and navigate to <http://192.168.99.101:8080>

We must see the same application as before

Try with the IP addresses of the other two nodes. It should work as well

Finally, we can stop the stack with

**docker stack rm bgapp**

And then, we can remove the secret

**docker secret rm db\_root\_password**

## Part 3: Podman

Make sure that all other machines (from the previous parts) are either deleted, or at least stopped

Navigate to the folder for **part 3**. This should be either **m3/almalinux/3**, or **m3/debian/3**

Start the VM

**vagrant up**

After the machine is up and running, open an SSH session to it

**vagrant ssh**

### Installation

As we are about to see, the manual installation of **Podman** is quite simple and you should not meet any difficulties automating it 😉

#### AlmaLinux OS

Execute the following

**sudo dnf -y install podman**

#### Debian

**sudo apt-get update**

**sudo apt-get -y install podman**

*Of course, it is always a good idea to upgrade all the installed packages*

### Exploration

We can start by asking for the version with

**podman --version**

Then, we can ask for the available commands

**podman --help**

We can recognize some of them (like **pull**, **run**, **images**, etc.) and the best part is that they are identical to their counterparts in **Docker**

#### Interactive Container

Let’s start a container, for example, an Ubuntu-based one

Use the following command

**podman run -it --name container1 ubuntu:24.04**

*Maybe you did notice that the image name we gave is considered as an alias to something else*

*You can check for details the* ***/etc/containers/registries.conf.d/000-shortnames.conf*** *file for AlmaLinux and the* ***/etc/containers/registries.conf.d/shortnames.conf*** *file for Debian*

Execute a command or two. For example, execute

**cat /etc/os-release**

After you are done, exit the session without terminating it – press **Ctrl+P** and then **Ctrl+Q**

Ask for the list of running containers with

**podman ps**

Ours is there. Now, we can see details about it by inspecting it

**podman inspect container1**

Let’s filter for something that we usually expect to be present

**podman inspect container1 | grep IPAddress**

It appears that **no IP address is assigned** to the container

There is a simple explanation. The container is running in **rootless mode**, thus no IP address

Let’s get back in the container

**podman attach container1**

Now check if there is an IP address in the container

**ip a**

If the command is missing, we can install it with

**apt update**

**apt install -y iproute2**

First, it appears that there is Internet access in the container

Second, after we have the **ip** command, we can see that the container has got an IP address

Let’s **exit** the container and terminate it

Check all containers including the running ones (not that we have any)

**podman ps -a**

Okay, our **container1** appears there as not running (as we expected)

#### Detached Container

Now, let’s run a container in detached mode

**podman run --name web1 -dt -p 8080:80/tcp httpd**

Depending on the OS and Podman’s version, we may see either an error that the short (image) name is not recognized, or we may be presented with a list to choose from

In order to avoid situations like this, we can always specify the full name

**podman run --name web1 -dt -p 8080:80/tcp docker.io/library/httpd**

Ask for the list of running containers with

**podman ps**

Ours is there. Now, we can see if we can reach the web application that is running inside the container

**curl http://localhost:8080**

Let’s ask for the container logs

**podman logs web1**

And then for the process running inside the container

**podman top web1**

We can also ask for stats on all the running containers (one in our case)

**podman stats**

Press **Ctrl+C** to stop the process

Now remove all the containers (running or not)

**podman rm --force $(podman ps -aq)**

#### Detached with a Volume Mount

Create a folder

**mkdir ~/nginx**

Create a simple index file there

**echo '<h2>It works (but on Podmap)!</h2' > ~/nginx/index.html**

Create and start a NGINX-based container that runs in detached mode and displays the recently created index file

**podman run --name web2 -dt -p 8080:80 -v ./nginx:/usr/share/nginx/html docker.io/library/nginx**

Ask for the list of running containers

**podman ps**

Ours is there. Now, we can see if we can reach the web application that is running inside the container

**curl http://localhost:8080**

And, depending on the host OS, you will see different results

On Debian, everything will behave as expected. And on AlmaLinux, an error message will appear stating “Permission denied”

The reason for this is that there is SELinux installed and running by default on all Red Hat-based distributions (including AlmaLinux OS)

In order to tackle this, when running a container with a mount, the command should be changed to

**podman run --name web2 -dt -p 8080:80 -v ./nginx:/usr/share/nginx/html:z docker.io/library/nginx**

Now the test should succeed even on AlmaLinux 😉

Remove the container

**podman rm --force web2**

#### Building Images

Let’s see how we can build a container image out of a file (either **Dockerfile** or **Containerfile**)

Prepare a **~/Containerfile** with the following content

**FROM docker.io/library/nginx:latest**

**COPY ./nginx/ /usr/share/nginx/html**

It is quite simple, and no surprises here besides the need to specify the full name of the base image

Now build the image

**podman build -t mynginx .**

List the images

**podman images**

It should be there

And finally, run a container out of it

**podman run --name web3 -dt -p 8080:80 mynginx**

Ask for the list of running containers

**podman ps**

Ours is there. Now, we can see if we can reach the web application that is running inside the container

**curl http://localhost:8080**

We should see it in all its glory 😉

Remove the container

**podman rm --force web3**

#### Podman Pods

Let’s explore a bit the new (at least for now) concept of pods

As we said earlier, a pod is a sort of grouping of containers that are closely related and connected. One popular example is a web service and a caching service

Besides the “user” containers, there is usually an infrastructure container as well. It prepares and holds the shared environment

We will use the **bgapp-web** and **bgapp-db** images to explore this topic (<https://github.com/shekeriev/bgapp>)

Let’s start by creating the pod first

**podman pod create --name bgapp-pod -p 8080:80**

And ask for the list of pods (just one should appear and it should be in created state)

**podman pod ps**

Now, let’s add the database container to the pod

**podman run -d --pod=bgapp-pod -e MYSQL\_ROOT\_PASSWORD=Parolka-12345 --name=db docker.io/shekeriev/bgapp-db**

And ask for the list of pods (just one should appear and it should be in running state)

**podman pod ps**

If you ask for the containers

**podman ps**

You will see two containers – the infrastructure one (pause) and our database container

**podman run -d --pod=bgapp-pod --name web docker.io/shekeriev/bgapp-web:embedded**

If you ask for the containers again

**podman ps**

You will see all the three containers – the infrastructure one (pause) and our web and database containers

Test if the application is reachable and working

**curl http://localhost:8080**

Yes, it should be 😊

Now, remove the whole pod (together with all its containers)

**podman pod rm --force bgapp-pod**

#### Docker Compose and Podman

First, we must make sure that our configuration has all the required components

One way to accomplish this is to install the **podman-compose** package

##### Preparation (AlmaLinux)

We need the **EPEL** repository first

**sudo dnf install -y epel-release**

And then we can install the required package

**sudo dnf install -y podman-compose**

##### Preparation (Debian)

Install the required package

**sudo apt-get install -y podman-compose**

##### An Example

Now, we can prepare a **compose.yaml** file with the following content

**services:**

**web:**

**image: docker.io/shekeriev/bgapp-web:embedded**

**ports:**

**- 8080:80**

**networks:**

**- app-network**

**depends\_on:**

**- db**

**db:**

**image: docker.io/shekeriev/bgapp-db:latest**

**networks:**

**- app-network**

**environment:**

**MYSQL\_ROOT\_PASSWORD: "${DB\_ROOT\_PASSWORD}"**

**networks:**

**app-network:**

First, let’s export a variable for the DB password

**export DB\_ROOT\_PASSWORD=Parolka-12345**

Start the composition with

**podman-compose up -d**

Even though you will see a report, you can always ask for the list of running containers

Either via **podman-compose**

**podman-compose ps**

Or through the usual command

**podman ps**

Finally check if the application is working as expected

**curl http://localhost:8080**

Yes, it should be 😊

Once done, remove all the services (containers)

**podman-compose down**

Check with

**podman ps -a**

Nothing left

However, the custom network is still there

**podman network ls**

Let’s remove it

**podman network rm vagrant\_app-network**

## Cleaning

All hosts (or instances) can be deleted either one by one or as a whole (from within the respective folder)

**vagrant destroy --force**