EXERCISE 1

Steps

1. **docker volume create ex1_vol**
   Creating a new docker volumes named ex1_vol. The volumes can be used to store stateful data from a container.

2. **docker run -v ex1_vol:/data -it –name sender ubuntu:22.04 bash**

   The command does multiple things. It is used to create a new container using the image ubuntu:22.04 from the docker registry(if not available in the local).
   The -v parameter is used to mount the /data directory in the newly created container to the ex1_vol docker volume.
   The new container is named as sender with –name parameter. The -it argument is used to start the container in interactive mode and attaches the current terminal to it, and the bash command at the end opens up a terminal of the sender container in the current terminal

3. **cd data, apt-get update && apt-get install vim, vim test.txt**

   Move inside the mounted directory and create a text file. Add in contents. For this exercise we enter “hello world!!”

4. **exit**

   This is used to exit the terminal of the sender container. This automatically exits the container as well. The container is stopped.

5. **docker run -v ex1_vol:/data -it –name receiver ubuntu:22.04 bash**

   Starting another container receiver in the same manner.

6. **cd data && ls && cat test.txt**

   This will print the text “hello world!!” in the terminal

EXERCISE 2

Steps and explanations. For the project setup, initially copy the install.sh file into a directory and create a new Dockerfile.

Dockerfile explanation

```
FROM ubuntu
COPY install.sh /
RUN cd /
RUN chmod +x install.sh
RUN ./install.sh
```

1. The first step is to get a base image. We use ubuntu as our base image here.
2. The next copies the shell script "install.sh" in the same directory and places it in the "/" of the newly created container
3. The next step is to change the directory of operation. We use RUN cd / to change the directory. This can also be achieved using the WORKDIR command in dockerfile.
4. We assign all executable permissions by using chmod +x . We are only granting executable permissions and not any read/write here.
5. The final step for building the image is running the shell script. This is done using the RUN command as well. All actions to be carried out in the shell for creating the image are done using RUN command

Commands executed

- **docker build -t ex2 .**
  This is run to build a docker image with tag ex2 from the dockerfile in the current directory.

- **Docker run -it ex2 bash**
  Start a new container from the docker image ex2 and start the container in interactive mode by attaching the current terminal to the docker container. The -i parameter starts the container in interactive mode and allows the user to pass arguments to the container. This is needed to provide inputs via command line. This also keeps the container alive for subsequent inputs. -t provides the terminal interface to see docker container outputs.

- **docker run -dit ex2**
  In this second mode of connecting to the containers terminal we start the containers with -dit argument. This -d stands for detached which means the container will run in the background and not in the foreground of the terminal from which it was started.
  The -d parameter enables the container to be run alive in the background.
  This will also print the resultant container id on the terminal.
  You can do a docker ps to verify that the container is indeed up and running.

- **docker attach $container_id**
  This is used to attach the current terminal to the docker container terminal. This gives access to the container's shell. It opens up the shell of the container for the user to type in any command.

EXERCISE 3: STARTING A PHP SERVER, PHP ADMIN AND MYSQL DB.

Dockerfile to start a PHP application server.

```
FROM php:8.0-apache
RUN docker-php-ext-install mysqli && docker-php-ext-enable mysqli
RUN apt-get update && apt-get upgrade -y
```

Description

1. From a base image of php:8.0 (which will be pulled from docker repository) we create a new docker image for our php server. The base image is a linux image image with php server configurations maintained.
2. In the second step we do the installation of mysql extensions using the docker-php-ext-install mysqli and docker-php-ext-enable mysqli commands. This enables the php server to communicate with sql servers by invoking functions. To run these commands when building the image we use the dockerfile RUN command.
3. We use dockerfile RUN command to update the system.

Docker-compose yml to build and start the 3 services

```
version: '3.9'
services:
    php-apache-environment:
        container_name: php-apache
        build:
            context: .
            dockerfile: Dockerfile
        depends_on:
            - db
        volumes:
            - ./php/src:/var/www/html/
        ports:
            - "8000:80"
    db:
        image: mysql
        container_name: db
        environment:
            MYSQL_ROOT_PASSWORD: MYSQL_ROOT_PASSWORD
            MYSQL_DATABASE: MYSQL_DATABASE
            MYSQL_USER: MYSQL_USER
            MYSQL_PASSWORD: MYSQL_PASSWORD
        ports:
            - "9906:3306"
        restart: always
    phpmyadmin:
        image: phpmyadmin/phpmyadmin
        container_name: admin
        depends_on:
            - db
        environment:
            PMA_HOST: db
        restart: always
        ports:
            - "9000:80"
```

We have 3 services in the above docker-compose.yaml

1. **php-apache-environment** : the PHP application server.
   Following are the important properties for the service in the yaml
   - Its build context is derived from the Dockerfile maintained in the same path.
   - It has a dependence on db service. This is added to ensure that the db service start prior to the application server.
   - There is a volume mounted from path ./php/source/ to /var/www/html of the container. This path if not present will be created on server startup. This path can be used to dynamically mount files to the container. In this case we start the container and copy our php source code into the path.
   - The php server is exposed at port 8000 of the local machine
2. **Db:** The MYSQL db server.
   Key properties
   - The base image is mysql downloaded from docker repository
   - Few environment variables have been set. These are used by the container to authenticate access to the db via ui,code etc.
   - DB server runs at port 3306 inside the container and exposed at port 9906 of the local machine
3. **phpmyadmin** : PHP Admin. The PHP admin provides a console to access the DB .
   - The service has a dependency on db and should always wait for the db server during startup.
   - The php admin needs the env variable name PMA_HOST to be set to the db uri. This is done by mentioning db as a service name and docker acts here as domain name resolver.
   - The PHP admin runs at port 9000. We can access the db via this. (localhost:9000)

Docker commands used to run the exercise

- **docker-compose up –build**
  The up command is used to start all the services mentioned in the docker-compose yml file. The –build option is used to rebuild the docker images if any changes had taken place. The steps builds the image and starts the container for all the services.

EXERCISE 4: KUBERNETES

K8 manifests used in this exercise.
Deployment

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello
spec:
  replicas: 2
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 100%
  selector:
    matchLabels:
      app: hello
  template:
    metadata:
      labels:
        app: hello
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions: [{ key: app, operator: In, values:
[hello] }]
            topologyKey: "kubernetes.io/hostname"
      containers:
      - name: hello-from
        image: pbitty/hello-from:latest
        ports:
          - name: http
            containerPort: 80
      terminationGracePeriodSeconds: 1
```

The deployment is a type of resource provided by k8s to manage the state of a running pod in k8s. In the above example we have a deployment yaml which has the following main properties.

- Deployment name is hello
- replicas: 2 This means whenever the deployment is created in the k8s there will be 2 instances of it. Basically this means 2 pods would be created.
- Update strategy is rolling update: this is the typical update strategy followed in k8s. When a new version is about to deployed not all the pods are replaced at once. This is to ensure zero downtime. When a deployment takes places, k8s creates a new replicase (which is another k8s resource). As soon as each new pod with new deployment version is ready one of the old pods is taken down and its replicaset value will be decremented. This maintains the total number pods and the availability of the system.
- There is a labelselector for app with labels “hello”.
- The container to be created during this deployment is from the image pbitty-hello-from:latest
- There is also a configuration for podAntiAffinity. This is used to make sure no two pods matching the criteria mentioned will end up in the same node. Opposite of this podAffinity.

Commands used to run the exercise

- **minikube start –nodes 2**
  Minikube is the CLI used to test and develop k8 applications in local. This command start a minikube cluster with 2 different nodes. In local this creates 2 different docker images.

- **kubectl get nodes**
  Gives a very brief overview of the nodes in the cluster, their ip, status etc.

- **kubectl apply -f hello-deployment.yml**
  The apply command is used to apply k8 resources to a cluster. Be it a pod, deployment, configmaps etc. The -f takes file name as argument. (in this case hello-deployment.yml)

- **Kubectl get pods -o wide**
  This gives info of pods in the cluster with detailed info on which nodes the pods, are running, its IP etc.

- **Kubectl delete deployment hello**
  Delete command is used to delete any resource type. In this case the resource type is deployment. It can be used to delete other resources like pods, replicasets etc. This is followed by the resource name.

- **Kubectl label nodes minikube_m02 disktype=kakashi**
  Labelling a node minikube_m02 with keyvalue “disktype=kakashi” . This can be used for node selection. Even if node identifiers change, the deployment manifests can be kept same with this.

Additional resource created

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello
spec:
  replicas: 2
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 100%
  selector:
    matchLabels:
      app: hello
  template:
    metadata:
      labels:
        app: hello
    spec:
      # affinity:
      #   podAntiAffinity:
      #     requiredDuringSchedulingIgnoredDuringExecution:
      #     - labelSelector:
      #         matchExpressions: [{ key: app, operator: In, values:
[hello] }]
      #       topologyKey: "kubernetes.io/hostname"
      containers:
      - name: hello-from
        image: pbitty/hello-from:latest
        ports:
          - name: http
            containerPort: 80
      nodeSelector:
        disktype: "kakashi"
      terminationGracePeriodSeconds: 1
```

The new deployment file is same as the old with little updates.

The hello app template has a new specification “**nodeselector: disktype: “kakashi”** “ so that the particular app is deployed only on those nodes The podAntiAffinity is also commented out so that 2 apps of label can indeed run on the same node.
If the podAntiAffinity is not removed then one of the pods will never be able to start.

- **Kubectl apply -f updated-deployment.yml**
  Deploying the new and updated deployment resource.