1. Run and Execute Any 6 Docker Commands and Comment on It

Theory:

Docker allows you to run applications in containers, ensuring consistency across different environments.

Here are **six fundamental Docker commands** to understand:

1. Build a Docker Image from a Dockerfile

This command is used to create a Docker image using a set of instructions defined in a `Dockerfile`. The image is built from the current directory where the `Dockerfile` is located.

- **Command**: `docker build -t [image_name] .`
- **What it does**: It creates an image with a name you specify, based on the instructions in the `Dockerfile`.

2. Run a Docker Container

This command runs a container from an image. The container will be executed in the background.

- **Command**: `docker run -d -p 8080:80 --name [container_name] [image_name]`
- **What it does**: It starts a container from an image and maps a port on your computer (8080) to the port inside the container (80). It also gives the container a name.

3. List All Containers (Running & Stopped)

This command shows all containers, whether they're running or stopped.

- **Command**: `docker ps -a`
- **What it does**: It lists all containers on your system, showing their status and details like names, IDs, and ports.

4. Stop a Running Container

This command stops a container that is currently running.

- **Command**: `docker stop [container_name]`
- **What it does**: It halts a running container, but the container still exists on your system.

5. Remove a Stopped Container

This command removes a container that is no longer running.

1. **Command**: `docker rm [container_name]`
2. **What it does**: It deletes a container that has been stopped from your system

—---

docker --version

Purpose: Checks the version of Docker installed on the system.

Explanation: Verifies which version of Docker you're using, ensuring compatibility with certain commands or features.

docker ps

Purpose: Lists only the running Docker containers.

Explanation: Displays container ID, name, status, and the ports exposed by running containers.

docker ps -a

Purpose: Lists all containers, including stopped ones.

Explanation: Shows containers that are no longer running but still exist on the system, useful for managing all containers.

docker images

Purpose: Displays a list of all images on the system.
Explanation: Lists available images with details like their repository name, size, and tags, so you know what you have to build containers.

docker run
Purpose: Creates and starts a container from an image.
Explanation: Runs a new container, for example, mapping a container's internal port to a host port and running it in detached mode.

docker stop
Purpose: Stops a running container.
Explanation: Halts a container from running, freeing up system resources. The container can still be restarted later.

---

2. Create a Docker File, Docker Image, and Run the Container

A **Dockerfile** automates the creation of Docker images. A **Docker image** is the environment that will be used to run a container.
Steps to Create a Dockerfile, Build an Image, and Run the Container:

1. **Create a Dockerfile**
    - A Dockerfile defines how to build a container image, including the base image, files to include, and the application's entry point.

Example Dockerfile:

```
 FROM nginx:alpine
COPY index.html /usr/share/nginx/html/index.html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

2. **Build the Image**
    - Once you've written the Dockerfile, use `docker build` to create the image.
        - Command: `docker build -t mynginx-image .`
    - **Explanation**: The `-t` flag tags the image with a name. The `.` refers to the directory where the Dockerfile is located.

3. **Run the Container**
    - To run a container from the image:
        - Command: `docker run -d -p 8080:80 mynginx-image`

    - **Explanation**: Starts the container in detached mode and maps port 80 inside the container to port 8080 on the host.

Explanation of Dockerfile Instructions:
- **FROM**: Sets the base image (here, Nginx with the Alpine version).
- **COPY**: Copies the `index.html` file from your local machine to the container.
- **EXPOSE**: Informs Docker that the container listens on port 80.
- **CMD**: Specifies the command to run when the container starts (in this case, starting the Nginx server).

---

3. Pull an Image from Docker Hub and Perform the Following Tasks
Theory:
Docker Hub is a registry that stores Docker images. You can pull pre-built images from there to quickly create containers.
Tasks:
1. **Pull an Image**
    - To get a Docker image, use `docker pull` followed by the image name.
        - Command: `docker pull nginx`
    - **Explanation**: Downloads the `nginx` image from Docker Hub. If no tag is specified, it defaults to the latest version.

2. **Create a Container**
    - After pulling an image, use `docker create` to prepare a container.
        - Command: `docker create --name mynginx-container nginx`
    - **Explanation**: Creates a container from the `nginx` image. The container is created but not started yet.

3. **Run the Container**
    - Run the container with the `docker run` command:
        - Command: `docker run -d -p 8080:80 mynginx-container`
    - **Explanation**: Starts the container in detached mode and maps port 8080 of the host to port 80 inside the container.

4. **Stop the Container**
    - To stop a running container, use the `docker stop` command:
        - Command: `docker stop mynginx-container`
    - **Explanation**: Stops the container, though it remains on the system for future use.

Why Docker Hub and Container Management?
- **Docker Hub** provides a large repository of pre-built images, saving you time in setting up containers.

- **Managing containers** (creating, running, stopping) is crucial for handling the lifecycle of applications running in containers.

—---

4. Create a Master Node and Worker Node in Kubernetes. Run Any 6 Commands of Kubernetes

Theory:

Kubernetes is an open-source platform for automating the deployment, scaling, and management of containerized applications. It uses a **master-worker architecture**, where the master node is responsible for the overall control of the cluster and the worker nodes run the actual applications (pods).

Master Node:

- The master node controls and manages the Kubernetes cluster. It is responsible for the scheduling of tasks and maintaining the overall health of the cluster.

- It manages the cluster's state, monitors, and coordinates activities across all the nodes (worker nodes).

- The master node contains several critical components:
  - **API Server**: Manages the Kubernetes API and handles REST requests.
  - **Controller Manager**: Manages controllers that regulate the state of the cluster (e.g., replication controllers).
  - **Scheduler**: Decides which node should run which pod.
  - **etcd**: A key-value store that holds the state of the cluster.

Worker Node:

- The worker node runs the actual application workloads. It contains the components required to run containers (pods).
- Each worker node has:
  - **kubelet**: Ensures that containers are running on the node.
  - **kube-proxy**: Manages network proxying and load balancing.
  - **Container Runtime**: Runs containers (e.g., Docker, containerd).

In the context of KillerCoda (or any Kubernetes setup), the master node and worker node are typically **pre-configured**. However, you can interact with them through the Kubernetes command-line tool `kubectl`.

6 Kubernetes Commands:

1. **Get Nodes:** This command shows the list of nodes in the cluster (both master and worker nodes).
   ```
   kubectl get nodes
   ```
2. **Get Pods:** Lists all the pods running in the Kubernetes cluster. Pods are the smallest deployable units in Kubernetes.
   ```
   kubectl get pods
   ```
3. **Create a Pod:** This command creates a pod from an image. For example, to create a pod using the `nginx` image:
   ```
   kubectl run my-pod --image=nginx
   ```
4. **Describe Pod:** To view detailed information about a specific pod (including events, resources, etc.), you can use the `describe` command:
   ```
   kubectl describe pod my-pod
   ```

5. **Expose Pod as a Service:** This command exposes a pod to the outside world by creating a service. You specify the port on which the service will listen:
   `kubectl expose pod my-pod --port=80 --type=NodePort`
6. **Get Services:** To check the services available in your cluster, use the following command:
   `kubectl get svc`

Understanding the Theory of Nodes and Commands:

- The **master node** manages the entire Kubernetes cluster. It is responsible for scheduling the pods onto the worker nodes, monitoring the health of the cluster, and scaling applications.
- **Worker nodes** run the containerized applications and services.
- Kubernetes commands like `kubectl get nodes`, `kubectl get pods`, and `kubectl expose` interact with the master and worker nodes to manage and deploy workloads.
- These commands allow you to inspect and control the cluster, deploy new applications, scale services, and monitor the state of the system.

—--

5. Create a Pod in Kubernetes, Find the IP Address, and Do the Troubleshooting via Logs

Theory:
In Kubernetes, a **Pod** is the smallest and simplest Kubernetes object. A pod represents a single instance of a running process in the cluster, and it encapsulates one or more containers, storage resources, and a unique network IP. When deploying applications in Kubernetes, they typically run inside pods.
Pod's Key Components:

- **Containers**: A pod can contain one or more containers that run together on the same node. Containers in the same pod share the same network IP, and hence can easily communicate with each other.
- **IP Address**: Each pod gets a unique internal IP address within the cluster, which it uses to communicate with other pods or services.
- **Volumes**: Pods can also define volumes to share data between containers.

Why Logs and Troubleshooting are Important:

- Logs are vital for debugging issues that arise in applications deployed in Kubernetes.
- Kubernetes provides an easy way to fetch logs using the `kubectl logs` command. Logs contain information like container status, errors, or output generated by the application.

Finding Pod IP and Logs:

- Once a pod is created, it will be assigned a unique IP address within the cluster's network. This is useful for inter-pod communication.
- If there are issues with the pod (such as containers not starting), you can troubleshoot by checking the logs or describing the pod for more detailed information.

Kubernetes Commands:

1.Create a Pod with a YAML File: To create a pod, you define its configuration in a YAML file (`mypod.yaml`). This file specifies details like the image to be used, ports to be exposed, and other configurations.

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mycontainer
    image: nginx
    ports:
    - containerPort: 80
```

After creating the YAML file, deploy the pod using:

2.**Get Pod IP:**

To find the IP address of a pod after deployment, you can use the following command:

```
kubectl get pod mypod -o wide
```

This will give you detailed information about the pod, including its IP address.

3.Get Pod Logs:

To check the logs of a pod and troubleshoot any issues, use:

```
kubectl logs mypod
```

This will show the output generated by the containers inside the pod.

4.Describe Pod:

For more detailed information about the pod, including its status, events, and resource usage, you can use:

```
kubectl describe pod mypod
```

This will provide insight into the pod's status and any issues with its deployment.

Theory of Pod Troubleshooting:

- **Creating a Pod**: The pod is the fundamental unit in Kubernetes for running containers. By creating a pod, you are deploying an application or service.
- **Finding the IP Address**: Each pod gets its own internal IP within the cluster. This helps in communication between pods. Pods can reach each other via their IP addresses, or services can be used to expose a pod externally.
- **Logs and Troubleshooting**: Logs are essential for debugging problems in applications. The `kubectl logs` command fetches logs from the containers within the pod, showing any output or errors they generate. This is especially useful when an application crashes or behaves unexpectedly.
- **Describing the Pod**: The `kubectl describe pod` command provides a comprehensive look at a pod's state, including resource usage, events, and errors. This helps diagnose issues with the pod, such as resource limits or misconfigurations.