10.11.25

Monday, November 10, 2025 3:17 PM

IAM Role

For example,
In this example, we will create an IAM role that permits EC2 instances read-only access to S3

The AWS CLI is an open-source tool that enables you to interact with AWS services from various command-line environments

For example, you can create an S3 bucket using the AWS CLI with the following command:
`aws s3api create-bucket --bucket my-bucket --region us-east-1`

After installation, you must configure the AWS CLI with your AWS account credentials. The configuration process includes providing your:

- AWS Access Key ID
- AWS Secret Access Key
- Default region for your resources
- Preferred output format (e.g., JSON, YAML, text, or table)

The credentials and settings will be stored in the hidden .aws

you might encounter two common issues:

1. Terraform may prompt for an AWS region. Although IAM resources are global, Terraform requires a region because most AWS resources are region-specific.
2. Terraform might not find valid AWS credentials to connect to your AWS account.

To address these issues, add a provider block to your configuration. The provider block specifies both the default region and the credentials needed to interact with your AWS account. The following combined configuration includes both the provider block and the IAM user resource block:

```
provider "aws" {
  region     = "us-west-2"
  access_key = "AKIAI44QH8DHBEXAMPLE"
  secret_key = "je7MtGbClwBF/2tk/h3yCo8n..."
}

resource "aws_iam_user" "admin-user" {
  name = "Lucy"
  tags = {
    Description = "Technical Team Leader"
  }
}
```

In this setup, the default region is set to US West 2. The access key and secret access key ensure Terraform can authenticate and make changes to your AWS account.

Hardcoding credentials in your Terraform configuration is not recommended, especially when storing files in version control.

AWS CLI Configuration

Configure the AWS CLI on your machine using:

```
aws configure
```

This creates a credentials file (typically located at `~/.aws/credentials`):

```
[default]
aws_access_key_id = YOUR_ACCESS_KEY_ID
aws_secret_access_key = YOUR_SECRET_ACCESS_KEY
```

Terraform will automatically use these stored credentials.

Environment Variables

Alternatively, you can set environment variables for your AWS credentials and region:

```
export AWS_ACCESS_KEY_ID=YOUR_ACCESS_KEY_ID
export AWS_SECRET_ACCESS_KEY=YOUR_SECRET_ACCESS_KEY
```

Environment Variables

Alternatively, you can set environment variables for your AWS credentials and region:

```
export AWS_ACCESS_KEY_ID=YOUR_ACCESS_KEY_ID
export AWS_SECRET_ACCESS_KEY=YOUR_SECRET_ACCESS_KEY
export AWS_DEFAULT_REGION=us-west-2
```

These methods enhance security by removing sensitive information from your Terraform configurations.

IAM Policies with Terraform

We will use the example of an IAM user named Lucy, who initially has no permissions.

Defining Resources in Terraform

Step 1: Declare the IAM User and IAM Policy

Below is a Terraform configuration snippet that first defines the IAM user resource, followed by the IAM policy resource:

```
resource "aws_iam_user" "admin-user" {
  name = "lucy"
  tags = {
    Description = "Technical Team Leader"
  }
}

resource "aws_iam_policy" "adminUser" {
  name   = "AdminUsers"
  policy = ?
}
```

Step 2: Incorporate the Policy Document with Heredoc Syntax

One efficient method to include the policy document within your Terraform configuration is to use a heredoc. This allows you to embed multi-line strings without external file references. Here's how to integrate the JSON document using this syntax:

```
resource "aws_iam_user" "admin-user" {
  name = "lucy"
  tags = {
    Description = "Technical Team Leader"
  }
}

resource "aws_iam_policy" "adminUser" {
  name   = "AdminUsers"
  policy = <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "*",
      "Resource": "*"
    }
  ]
}
EOF
}
```

Step 3: Attaching the Policy to the IAM User

Even though the IAM policy is defined, it is not automatically granted to Lucy. To attach the policy, we use the `aws_iam_user_policy_attachment` resource. This resource takes the username and the ARN of the IAM policy as inputs:

```
resource "aws_iam_user_policy_attachment" "lucy-admin-access" {
  user       = aws_iam_user.admin-user.name
  policy_arn = aws_iam_policy.adminUser.arn
}
```

Introduction to AWS S3

Data in S3 is organized into containers called ***buckets***. Each bucket can hold an unlimited number of objects, and every file stored is treated as a separate object—even when they appear to be organized within folders such as "pictures/cat.jpg" or "videos/dog.mp4".

Once created, the bucket is accessible via a unique DNS endpoint. For example, a bucket named "allpets" in the US West (N. California) region would be accessible at: https://allpets.us-west-1.amazonaws.com

Bucket policies are JSON documents that control access to your S3 buckets. They can grant or restrict permissions for IAM users, groups, or even external accounts.

Below is an example policy that allows an IAM user named Lucy to retrieve all objects from a bucket called "all-pets":

```
{
    "Version": "2012-10-17",
```

Below is an example policy that allows an IAM user named Lucy to retrieve all objects from a bucket called "all-pets":

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "s3:GetObject"
            ],
            "Effect": "Allow",
            "Resource": "arn:aws:s3:::all-pets/*",
            "Principal": {
                "AWS": [
                    "arn:aws:iam::123456123457:user/Lucy"
                ]
            }
        }
    ]
```

Creating an S3 Bucket

To create an S3 bucket, we use the AWS S3 bucket resource in Terraform. For more details on the available resource arguments, please refer to the Terraform AWS documentation.

Below is an example configuration where we define an S3 bucket with a unique name and attach a descriptive tag.

```
resource "aws_s3_bucket" "finance" {
  bucket = "finance-21092020"
  tags = {
    Description = "Finance and Payroll"
  }
}
```

Below is an example configuration for uploading a file to your bucket:

```
resource "aws_s3_bucket_object" "finance-2020" {
  content = "/root/finance/finance-2020.doc"
  key     = "finance-2020.doc"
  bucket  = aws_s3_bucket.finance.id
}
```

What is a data block in Terraform?

- A **data block** is used to **read or fetch existing resources** from your cloud provider (AWS in this case).
- It does **NOT create** anything. It only **retrieves information** about something that already exists.
- Example: If you already have an IAM group in AWS and you want to use its ARN or ID in your Terraform code, you use a data block.

Your example:

```
data "aws_iam_group" "finance-data" {
```

group_name = "finance-analysts"
}

Show more lines

- **data "aws_iam_group"** → This tells Terraform: “Fetch details of an existing IAM group from AWS.”
- **finance-data** → This is the **local name** inside Terraform. You use it to reference this data later in your code. Example: data.aws_iam_group.finance-data.arn
- **group_name = "finance-analysts"** → This is the **actual IAM group name in AWS** that you want to look up.

Difference between resource and data:

- resource → **Creates** something new in AWS (e.g., an S3 bucket, IAM group).
- data → **Reads** something that already exists in AWS (e.g., an existing IAM group).

bucket policy attached to granting full access to the specified IAM entity

Introduction to DynamoDB

Amazon DynamoDB, AWS's fully managed NoSQL database solution.

Its data replication capability across multiple AWS regions further guarantees high availability, a feature trusted by many large-scale applications.

DynamoDB organizes data using key-value pairs and document structures.

You might begin with a table that includes keys such as manufacturer and model. As your data

requirements evolve, you can enhance the table by adding keys like the manufacturing year and the vehicle identification number (VIN).

Each row in the table is referred to as an "item."


```
{
 "Manufacturer": "Toyota",
 "Make": "Corolla",
 "Year": 2004,
 "VIN": "4Y1SL65848Z411439"
}
{
 "Manufacturer": "Honda",
 "Make": "Civic",
 "Year": 2017,
 "VIN": "DY1SL65848Z411432"
}
{
 "Manufacturer": "Dodge",
 "Make": "Journey",
 "Year": 2014,
 "VIN": "SD1SL65848Z411443"
}
{
 "Manufacturer": "Ford",
 "Make": "F150",
 "Year": 2020,
 "VIN": "DH1SL65848Z411100"
}
```


Here, each of the block is an item.

creation and the insertion of a single item into the table

```
resource "aws_dynamodb_table" "cars" {
  name         = "cars"
  hash_key     = "VIN"
  billing_mode = "PAY_PER_REQUEST"
  attribute {
    name = "VIN"
    type = "S"
  }
}

resource "aws_dynamodb_table_item" "car-items" {
  table_name = aws_dynamodb_table.cars.name
  hash_key   = aws_dynamodb_table.cars.hash_key
  item       = <<EOF
{
  "Manufacturer": {"S": "Toyota"},
  "Make": {"S": "Corolla"},
  "Year": {"N": "2004"},
  "VIN": {"S": "4Y1SL65848Z411439"}
}
EOF
}
```

1. aws_dynamodb_table "cars"

- This resource **creates the DynamoDB table** named cars.
- It defines:
  - Table name (cars)
  - Primary key (hash_key = "VIN")
  - Billing mode (PAY_PER_REQUEST)
  - Attribute schema (VIN as a string)

So this resource is about **creating the structure** of the database.

2. aws_dynamodb_table_item "car-items"

- This resource **inserts an item (a record)** into the table you just created.
- It uses:
  - table_name = aws_dynamodb_table.cars.name → Links to the table created above.
  - hash_key = aws_dynamodb_table.cars.hash_key → Ensures the item matches the table's primary key.
  - item → The actual data for the record:

JSON
{
"Manufacturer": {"S": "Toyota"},
"Make": {"S": "Corolla"},
"Year": {"N": "2004"},
"VIN": {"S": "4Y1SL65848Z411439"}
}
Show more lines

- S = String type
- N = Number type

So this resource is about **adding data to the table**.

☑ **Why two resources?**

- **One resource defines the table (schema)**.
- **Another resource adds items (data)**. Terraform separates these because:
- You might create the table once but insert many items later.
- It keeps infrastructure (table) and data (items) modular.

Challenges with Local State Files

In early configurations, the state file was created and maintained on a developer's machine.

Imagine a scenario where a developer named Abdul creates a Terraform configuration for provisioning an S3 bucket. After running terraform plan and applying the configuration, Terraform generates a local state file (terraform.tfstate). Abdul then commits all configuration files, including the state file, into a Git repository. Later, when another developer, Lee, pulls the repository, he makes his modifications, reviews the plan, applies the changes, and pushes the updated configuration and state file back to the Git repository

This process, while seemingly functional, introduces significant risks:

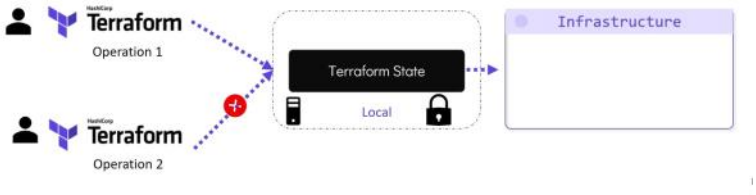
- Sensitive infrastructure details (such as IP addresses and key names) are stored within a Git repository.
- Concurrent updates using local state files can lead to conflicts or even state corruption.

How Terraform State Locking Works

Terraform incorporates a mechanism called state locking to prevent simultaneous modifications. When you run commands like terraform apply, Terraform locks the state file to avoid interference from another operation.

Terraform acquires a state lock to protect the state from being written by multiple users at the same time.

State Locking



But there's a problem.

working with outdated state files (by not pulling the latest changes) can lead to accidental destructive actions, such as unintended resource rollbacks or deletions.

Also, Version control systems like GitHub do not support **state locking.**

Remote Backends: A Secure Alternative

Remote backends store the state file outside the configuration directory and version control system using services such as AWS S3, Google Cloud Storage ect.

With a remote backend, Terraform automatically:

- Loads the state from shared storage for every operation.
- Uploads state updates after each terraform apply.
- Provides state locking to maintain state integrity.

Using remote backends ensures that your Terraform state is managed centrally and securely, supporting team collaboration

When you run terraform apply, Terraform initially generates a local state file (terraform.tfstate).
To switch to a remote backend, add a Terraform block with backend settings.

```
resource "local_file" "pet" {
  filename = "/root/pets.txt"
  content  = "We love pets!"
}

terraform {
  backend "s3" {
    bucket         = "kodekloud-terraform-state-bucket01"
    key            = "finance/terraform.tfstate"
    region         = "us-west-1"
    dynamodb_table = "state-locking"
  }
}
```

Breakdown of the Configuration

- bucket: The name of the S3 bucket where the state file will be stored.
- key: The S3 object path for storing the state file (in this example, under a folder named "finance").
- region: The AWS region where your S3 bucket is located.
- dynamodb_table: Name of the pre-created DynamoDB table enabling state locking to ensure safe concurrent updates.

Then run terraform init.

`$ rm -rf terraform.tfstate ->` To ensure it only runs server state, not local

Terraform State Commands

The state file is stored in JSON format, and it is crucial not to edit it manually.

$ terraform state list -> all resource addresses

What is a Terraform Provisioner?

- A **provisioner** in Terraform is used to **execute scripts or commands on a resource after it is created or destroyed**.
- Think of it as a way to **configure the resource beyond just creating it**.

Why do we need provisioners?

- Sometimes, after creating infrastructure (like an EC2 instance), you need to:
  - Install software
  - Run configuration scripts
  - Copy files
- Provisioners help automate these tasks **within Terraform**.

Types of Provisioners

1. **local-exec**
   - Runs a command **on your local machine** (where Terraform runs).
     Example:

     Terraform
     provisioner "local-exec" {
      command = "echo Instance created!"
     }
     Show more lines
2. **remote-exec**
   - Runs commands **on the remote resource** (e.g., an EC2 instance) via SSH or WinRM.
     Example:

     provisioner "remote-exec" {
      inline = [
      "sudo apt-get update",
      "sudo apt-get install nginx -y"
      ]
     }
3. **file**
   - Copies files from your local machine to the remote resource.
     Example:

     provisioner "file" {
      source = "app.conf"
      destination = "/etc/app.conf"

When to use provisioners?

- **Last resort**: Terraform recommends using **cloud-init, user data, or configuration management tools (Ansible, Chef)** instead.

creating an AWS EC2 instance with Terraform:

☑ Basic Steps

1. **Provider Configuration**
   Tell Terraform you're using AWS:

   ```
   Terraform
   provider "aws" {
    region = "us-east-1"
   }
   ```
   Show more lines
2. **EC2 Resource Block**
   Define the EC2 instance:

   ```
   Terraform
   resource "aws_instance" "my_ec2" {
    ami = "ami-0c55b159cbfafe1f0" # Amazon Linux AMI
    instance_type = "t2.micro"
    key_name = "my-key" # SSH key pair name
    tags = {
    Name = "MyEC2Instance"
    }
   }
   ```
   Show more lines

☑ Do we need SSH configured?

- **Yes, if you want to connect to the EC2 instance** after it's created.
- You need:
  - An **AWS Key Pair** (created in AWS console or via Terraform).
  - The **private key (.pem)** on your local machine.
- Terraform does **not automatically create SSH keys** unless you define them:

  ```
  Terraform
   resource "aws_key_pair" "my_key" {
   key_name = "my-key"
   public_key = file("~/.ssh/id_rsa.pub")
   }
  ```
  Show more lines
- Then reference key_name = aws_key_pair.my_key.key_name in your EC2 resource.

☑ Why SSH matters?

- Without SSH, you **cannot log in** to the instance for configuration or troubleshooting.
- If you plan to use **provisioners (remote-exec)**, SSH is mandatory because Terraform needs to connect to the instance.

☑ What is Terraform Taint?

- **Terraform taint** is a command that **marks a resource for recreation** during the next terraform apply.
- It does **not delete immediately**, but tells Terraform:
  "This resource is broken or needs to be replaced, so destroy and recreate it."

```
terraform taint aws_instance.my_ec2
```

- This marks aws_instance.my_ec2 for recreation.
- Next terraform apply will **destroy and recreate** that EC2 instance.

Why use environment variables in Terraform?

Environment variables let you **control Terraform behavior without changing your `.tf` files**

Common ones:

Variable	Purpose
`TF_LOG`	Set log level (`TRACE`, `DEBUG`, `INFO`)
`TF_LOG_PATH`	Save logs to a file
`TF_VAR_<name>`	Pass values to Terraform variables

Example:

```
export TF_LOG=DEBUG
export TF_LOG_PATH=/tmp/terraform.log
export TF_VAR_region="us-east-1"
```

What does this mean?

```
export TF_LOG_PATH=/tmp/terraform.log
```

- `export` → Sets an **environment variable** in your shell.
- `TF_LOG_PATH` → A special Terraform variable that tells Terraform **where to write logs**.
- `/tmp/terraform.log` → The file where logs will be saved.

So, instead of showing logs on the terminal, Terraform will write them to `/tmp/terraform.log`.

☑ What is terraform import?

- terraform import is a command that **brings an existing resource in your cloud (AWS, Azure, etc.) under Terraform management**.
- It does **not create** the resource; it just **adds it to Terraform state** so Terraform can track it.

☑ Why do we need it?

- If you already have resources created manually (e.g., an EC2 instance in AWS console) and you want Terraform to manage them **without recreating**.
- It helps in **migrating existing infrastructure to Terraform**.

✅ How does it work?

1. You **write the resource block** in your `.tf` file (matching the resource type and name). Example:

```terraform
resource "aws_instance" "my_ec2" {
  # No need to fill all details yet
}
```

2. Run the import command:

```shell
terraform import aws_instance.my_ec2 i-0abcd1234efgh5678
```

   - `aws_instance.my_ec2` → Terraform resource address.
   - `i-0abcd1234efgh5678` → The actual AWS EC2 instance ID.

3. Terraform updates its **state file** to include this resource.

terraform import = **Attach existing resource to Terraform state so Terraform can manage it**.

✅ Structure of a Module

A module is just a folder with `.tf` files:

```
my-module/
  main.tf
  variables.tf
  outputs.tf
```

- **`main.tf`** → Contains resources.
- **`variables.tf`** → Defines input variables.
- **`outputs.tf`** → Defines outputs for other modules or root configuration.

Step 1: Define the module (in `modules/ec2-instance`)

```hcl
# modules/ec2-instance/main.tf
resource "aws_instance" "this" {
  ami           = var.ami
  instance_type = var.instance_type
}
```

Step 2: Call the module

```hcl
module "web_server" {
  source         = "./modules/ec2-instance"
  ami            = "ami-0c55b159cbfafe1f0"
  instance_type  = "t2.micro"
}
```

Below is an example of how you might structure your Terraform files across multiple files:

```
# main.tf
resource "aws_instance" "webserver" {
  # configuration here
}
```

```
# key_pair.tf
resource "aws_key_pair" "web" {
  # configuration here
}
```

```
# dynamodb_table.tf
resource "aws_dynamodb_table" "state-locking" {
  # configuration here
}
```

```
# security_group.tf
resource "aws_security_group" "ssh-access" {
  # configuration here
}
```

```
# ec2_instance.tf
resource "aws_instance" "webserver-2" {
  # configuration here
}
```

```
# s3_bucket.tf
resource "aws_s3_bucket" "terraform-state" {
  # configuration here
}
```

Consider the following directory listing for a typical Terraform project:

```
$ ls
provider.tf
id_rsa
id_rsa.pub
main.tf
pub_ip.txt
terraform.tfstate.backup
terraform.tfstate
iam_roles.tf
iam_users.tf
security_groups.tf
variables.tf
outputs.tf
s3_buckets.tf
dynamo_db.tf
local.tf
```

☑ What is Terraform Registry?

- It's a public repository of **pre-built Terraform modules** for AWS, Azure, GCP, Kubernetes, etc.
- URL: https://registry.terraform.io

Terraform Registry modules = **Ready-made building blocks for common infrastructure tasks**.

We needed to confirm that all the different services worked seamlessly with our chosen OS version (a Node.js web server, a MongoDB database, a Redis messaging system, and an orchestration tool like Ansible.)

For new members in any project, Every new developer had to follow an extensive setup process and execute numerous commands to ensure their environment was correctly configured

1. Correct OS
2. Proper version of each component

With Docker, every component runs inside its own container with dedicated dependencies and libraries, all on the same virtual machine and operating system while remaining isolated from each other.

Containers are isolated environments that operate independently from one another

Each container maintains its own processes, network interfaces, and mounts—similar to virtual machines—but all containers share the host OS kernel.

> For instance, if you have an Ubuntu system with Docker installed, you can also run containers based on Debian, Fedora, SuSE, or CentOS because they all utilize the same Linux kernel. It is crucial to understand that Docker leverages the Docker host's kernel.

Although many users have installed Docker on Windows to run Linux containers, these Linux containers operate within a Linux virtual machine under the hood.

Docker Image can be of OS, database, tools etc.

```
docker run ansible
docker run mongodb
docker run redis
docker run nodejs
docker run nodejs
```

A Docker image is a pre-built package or template (similar to a virtual machine template) used to create one or more container instances. Containers, on the other hand, are the actual running instances of these images, each offering its own isolated environment and processes.

If you cannot locate a pre-existing image that meets your requirements, you can build your own Docker image and push it to repositories like Docker Hub

developers and operations can collaborate using a Dockerfile that encapsulates all configuration details

Docker simplifies containerization by allowing you to build, ship, and run applications consistently across various environments

Installing Docker

Before installing Docker, it is crucial to remove any existing Docker packages. Run the following command:

```
sudo apt-get remove docker docker.io docker-engine
```

When you install Docker using the package manager:

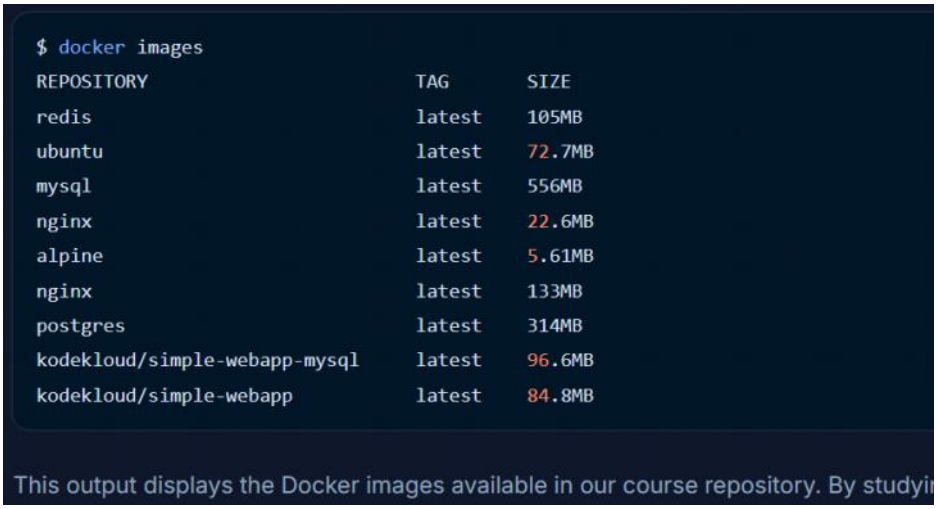
- You add Docker's official repository to your system.
- You download verified packages from that repository.
- Updates are handled automatically with system updates (apt-get upgrade).

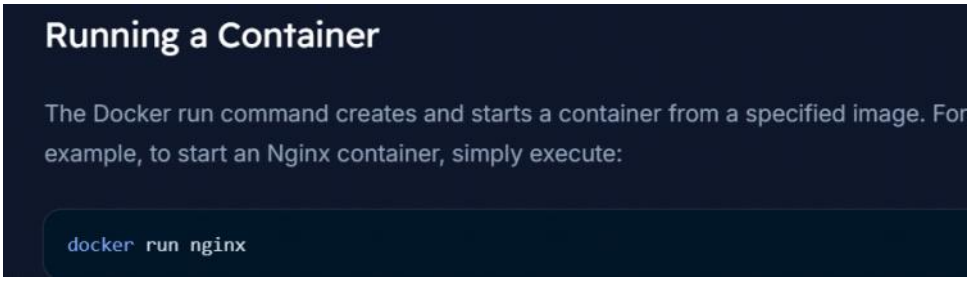
A **convenience script** is a pre-written shell script provided by Docker that automates the installation process.
This script:

- Detects your OS.
- Installs Docker without you manually adding repositories or keys.

To confirm Docker is running correctly, try launching a simple container.

```
docker run docker/whalesay cowsay boo
```

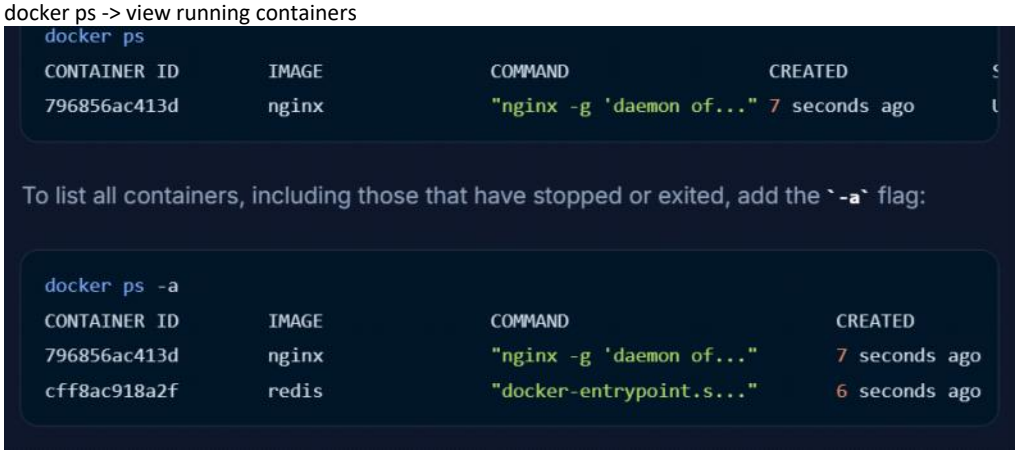




However, if the image is not present, Docker will pull it from Docker Hub.
if you want to download an image for later use without running a container immediately, use: docker pull"

docker pull nginx

docker ps -> view running containers



To stop a running container, provide the container ID or name.
docker stop silly_sammet

Stopping and Removing Containers

To stop a running container, provide the container ID or name. First, confirm the container details with:

```
docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS         PORTS
796856ac413d   nginx     "nginx -g 'daemon of..." 7 seconds ago   Up 6 seconds   80/tcp
```

Then, stop the container by running:

```
docker stop silly_sammet
```

After stopping, running `docker ps` will show no active containers. To permanently remove a stopped container, use:

```
docker rm silly_sammet
docker ps
```

Docker ps -> running containers
Docker images -> images present locally

Managing Docker Images

Viewing local Docker images is straightforward with the `docker images` command. This command displays each image along with its size, creation time, and more:

```
docker images
REPOSITORY          TAG       IMAGE ID       CREATED         SIZE
nginx               latest    f68d6e55e065   4 days ago      109MB
redis               latest    4760dc956b2d   15 months ago   107MB
ubuntu              latest    f975c503748    16 months ago   112MB
alpine              latest    3fd9065eaf02   18 months ago   4.14MB
```

docker rmi nginx -> To delete an image no longer needed.

Running ubuntu containers is absurd.
docker run ubuntu ->

will start a container that immediately exits. This occurs because Ubuntu, by default, has no long-running process.

To keep ubuntu container active ->
`docker run ubuntu sleep 5`

docker exec -> run commands inside a running container without starting a new one.

- **Detached Mode (`-d`):**

```
docker run -d kodekloud/simple-webapp
```

  - The container runs **in the background**.
  - Your terminal is free for other commands.
  - Docker prints the **container ID** instead of logs.

```
docker run -d -p 8080:8080 kodekloud/simple-webapp
```

This runs the app in the background and makes it accessible on `http://localhost:8080`.

If you need to reattach to the container, use the **`docker attach`** command along with the container ID (a shortened unique prefix is acceptable):

```
docker attach a043d
```

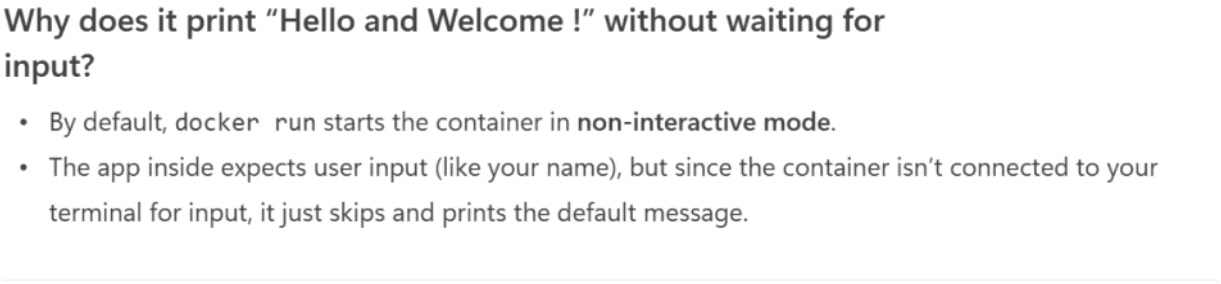
docker run redis:4.0 -> if u don't want latest

kodekloud/simple-prompt-docker is a Docker image that contains the application you described.
Here's what's happening:

- The image kodekloud/simple-prompt-docker is hosted on **Docker Hub** under the kodekloud namespace.
- When you run:

docker run kodekloud/simple-prompt-docker
Docker:

1. Pulls the image from Docker Hub (if not already present locally).
2. Creates a container from that image.
3. Executes the default command inside the container (which runs the app).



✅ To make it interactive, you need to use:

```
1  docker run -it kodekloud/simple-prompt-docker
```

- `-i` → Keeps STDIN open.
- `-t` → Allocates a pseudo-terminal. Together, `-it` lets you interact with the app inside the container.

i -> interactive
t -> terminal

docker run kodekloud/webapp
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)

docker run -p 80:5000 kodekloud/webapp

Here,

Port 80 vs Port 5000

- **Port 5000**
  - This is the **port inside the container** where the web application is listening.
  - When you run docker run kodekloud/webapp, the app starts and says:
    * Running on http://0.0.0.0:5000/

    That means **inside the container**, the app is available on port 5000.
- **Port 80**
  - This is the **port on your host machine** (your laptop/server) that you want to expose to the outside world.
  - Port 80 is the **default HTTP port**, so if you map it, users can access the app by visiting http://localhost without specifying a port.

Now, users can access your web application via http://192.168.1.5:80 (assuming 192.168.1.5 is the Docker host IP)

container filesystems are ephemeral, all data is lost when the container is removed. -> so we need data persistence

By default, a MySQL container stores its data in /var/lib/mysql inside the container. But data is lost when container is removed.

docker run mysql
docker stop mysql
docker rm mysql
Here all data is removed from container, as container is removed.

To persist the mysql container data ,
create a directory /opt/datadir on your host (not inside docker container, rather my laptop) and run: (name of directory can be different)

docker run -v /opt/datadir:/var/lib/mysql mysql
This volume mapping ensures that the data in /opt/datadir remains intact even if the container is deleted.

docker inspect -> for detailed info about running container (since docker ps gives less info).

docker logs blissful_hopper -> to view logs from container named “blissful_hopper,

Command:

docker run -p 8080:8080 -p 50000:50000 -v /root/my-jenkins-data:/var/jenkins_home -u root jenkins

What each part means:

1. **docker run**
   Starts a new container from the specified image (jenkins in this case).

2. **-p 8080:8080 -p 50000:50000**
   Port mapping:
   - 8080:8080 → Maps **host port 8080** to **container port 8080**.
     Jenkins web UI runs on port 8080 inside the container, so you can access it via http://<host-ip>:8080.
   - 50000:50000 → Maps **host port 50000** to **container port 50000**.
     Jenkins uses this for **agent communication** (when connecting build agents to Jenkins).

3. **-v /root/my-jenkins-data:/var/jenkins_home**
   Volume mapping:
   - Maps the host directory /root/my-jenkins-data to the container directory /var/jenkins_home.
   - Jenkins stores all its configuration, jobs, and plugins in /var/jenkins_home.
     This ensures **data persistence** even if the container is removed.

4. **-u root**
   Runs the container as the **root user** instead of the default Jenkins user.
   This is sometimes needed for permission issues when mounting volumes.

5. **jenkins**
   This is the **Docker image name** (official Jenkins image from Docker Hub).

What does this achieve?

- Starts Jenkins in a container.
- Makes Jenkins UI accessible on http://localhost:8080.

- Allows Jenkins agents to connect via port 50000.
- Persists Jenkins data on your host machine.
- Runs as root for permissions.

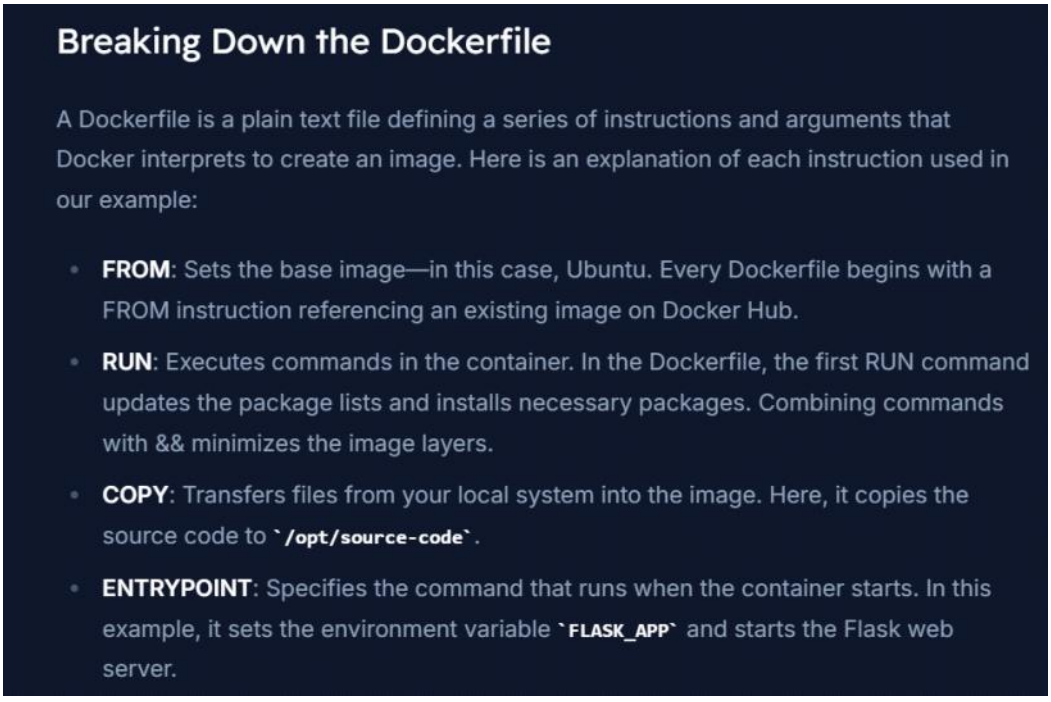
☑ **In short:**
This command runs Jenkins in a container with **port mapping**, **persistent storage**, and **root privileges**.

Creating my own docker image

Dockerfile that encapsulates the entire process.
Create a dockerfile by add instructions for setting up your application by installing dependencies, copying the source code, and setting the entrypoint.

```
FROM ubuntu

RUN apt-get update && apt-get install -y python python-pip

RUN pip install flask flask-mysql

COPY . /opt/source-code

ENTRYPOINT ["sh", "-c", "FLASK_APP=/opt/source-code/app.py flask run"]
```

Now dockerfile is in place, build your image locally by -> docker build



COPY . /opt/source-code

- Copies everything from your **current directory** (where the Dockerfile is) into /opt/source-code inside the container.
- This is where your application code will live.

5. ENTRYPOINT ["sh", "-c", "FLASK_APP=/opt/source-code/app.py flask run"]

- **ENTRYPOINT** defines the command that runs when the container starts.
- Here:
  - FLASK_APP=/opt/source-code/app.py → Tells Flask which app to run.
  - flask run → Starts the Flask development server.
- ["sh", "-c", "..."] means run the command in a shell.

☑ **What does this Dockerfile do overall?**

- Creates an image based on Ubuntu.
- Installs Python and Flask.
- Copies your app code into the image.
- Runs the Flask app when the container starts.

The **Dockerfile is always on your local machine**, not inside the container.
Here's why:

- **Purpose of Dockerfile**:
  It's a set of instructions for Docker to **build an image**.
  You write it locally, then run:

Shell

docker build -t my-image .
Show more lines
Docker reads the Dockerfile and creates an image.

- **When does the container come into play?**
  After the image is built, you start a container from that image using:

Shell
docker run my-image
Show more lines
At this point, the container is created **from the image**, which was built using the Dockerfile.

☑ **In short:**

- **Dockerfile** → Local machine (used during image build).
- **Container** → Runs after the image is built (doesn't contain the Dockerfile unless you copy it explicitly).

Popular applications containerized with Docker include web browsers like Chrome and Firefox, utilities like cURL, and applications like Spotify or Skype. (apart from web apps)

```python
import os
from flask import Flask, render_template

app = Flask(__name__)
color = os.environ.get('APP_COLOR')

@app.route("/")
def main():
    print(color)
    return render_template('hello.html', color=color)

if __name__ == "__main__":
    app.run(host="0.0.0.0", port="8080")
```

Running the Application with an Environment Variable

We will be setting color, by setting the environment variable inline.

```
export APP_COLOR=blue; python app.py
```

For example, to run the container with the background color set to blue, execute:

docker run -e APP_COLOR=blue simple-webapp-color

To deploy multiple containers with different background colors, you can run multiple containers with distinct values:

docker run -e APP_COLOR=blue simple-webapp-color
docker run -e APP_COLOR=green simple-webapp-color
docker run -e APP_COLOR=pink simple-webapp-color

Unlike virtual machines, containers are designed to run a specific task or process (e.g., hosting a web server, application server, or database). Once that task completes or the process crashes, the container stops running.

A container runs **only as long as its main process is running**.

What does this Dockerfile do?

Dockerfile

```
1  FROM ubuntu
2  CMD ["sleep", "5"]
```

- When you run docker run <image>, the container will **pause for 5 seconds** and then exit.
- CMD is **not executed during build**, only when the container runs.

Why are we using sleep in the Dockerfile?

- A container runs **only as long as its main process is running**.
  If you start a container from ubuntu without specifying a command:

  Shell
  docker run ubuntu
  Show more lines
  It will **start and immediately exit**, because there's no long-running process.
- Adding CMD ["sleep", "5"] means:
  - When the container starts, it runs sleep 5 (pauses for 5 seconds).
  - After 5 seconds, the command finishes → container stops.

What if we don't add sleep?

- The container will **exit immediately** after starting because there's nothing to keep it alive.
- Docker containers are designed to run a process. If that process ends, the container ends.

How do we keep a container running?

- Use a long-running process like:
  - A web server (nginx, flask, etc.)
  - Or an infinite sleep:

    Dockerfile
    CMD ["sleep", "infinity"]
    Show more lines
    This keeps the container alive forever until you stop it manually.

Making the Change Permanent” using CMD

Here’s the logic behind **“Making the Change Permanent”**:

Before adding CMD

If you run:

Shell
docker run ubuntu sleep 5
Show more lines

- You are telling Docker:
  “Start a container from the ubuntu image and run sleep 5 inside it.”
- This works, but you have to **type sleep 5 every time** you start the container.

After adding CMD in Dockerfile

Dockerfile
FROM ubuntu
CMD ["sleep", "5"]
Show more lines

- You build a new image:

  Shell
  docker build -t my-ubuntu-sleeper .
  Show more lines
- Now, when you run:

  Shell
  docker run my-ubuntu-sleeper
  Show more lines
  Docker automatically runs sleep 5 because **CMD sets the default command** for the image.

☑ **Why is this useful?**

- It makes the behavior **permanent** for that image.
- You don’t need to remember or type the command every time.
- You can still override it if needed:

  Shell
  docker run my-ubuntu-sleeper sleep 10
  Show more lines
  (This replaces the default sleep 5 with sleep 10.)

Why sleep at all?

- Containers stop when their main process ends.
- If you start an ubuntu container without a command, it exits immediately.
- Adding sleep keeps the container alive for testing or debugging.

- In real apps, the main process would be your web server, database, etc.

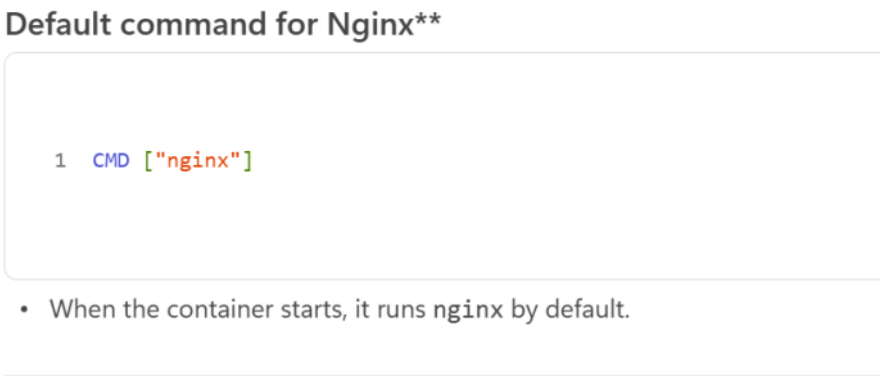
The **WORKDIR** instruction in a Dockerfile always refers to a **directory inside the container's filesystem**, not your local machine.

☑ What happens with WORKDIR?

- When you write:

Dockerfile
WORKDIR /etc/nginx
Show more lines
Docker sets the **current working directory inside the container** to /etc/nginx.

- Any subsequent instructions like RUN, CMD, ENTRYPOINT, COPY, or ADD will execute relative to this directory.
- If /etc/nginx doesn't exist, Docker creates it during the image build.



Here's a **lucid explanation of the Dockerfile** you shared:

1. Install Nginx

Dockerfile
RUN \
 add-apt-repository -y ppa:nginx/stable && \
 apt-get update && \
 apt-get install -y nginx && \
 rm -rf /var/lib/apt/lists/* && \
 echo "\ndaemon off;" >> /etc/nginx/nginx.conf && \
 chown -R www-data:www-data /var/lib/nginx
Show more lines

- Adds the Nginx repository and installs Nginx.
- Cleans up package lists to reduce image size.

- Adds daemon off; to Nginx config so it runs in **foreground** (important for Docker).
- Changes ownership of Nginx files to www-data.

2. Define mountable directories

Dockerfile
VOLUME ["/etc/nginx/sites-enabled", "/etc/nginx/certs"]
Show more lines

- Declares directories that can be mounted from the host.
- Useful for custom Nginx configs and SSL certificates.

3. Set working directory

Dockerfile
WORKDIR /etc/nginx
Show more lines

- Inside the **container**, this becomes the default directory for subsequent commands.

Default command for Nginx**

Dockerfile
CMD ["nginx"]
Show more lines

- When the container starts, it runs nginx by default.

5. Install MySQL

Dockerfile
RUN rpmkeys --import https://repo.mysql.com/RPM-GPG-KEY-mysql \
 && yum install -y $MYSQL_SERVER_PACKAGE_URL $MYSQL_SHELL_PACKAGE_URL libpwquality \
 && yum clean all \
 && mkdir /docker-entrypoint-initdb.d
Show more lines

- Imports MySQL GPG key for package verification.
- Installs MySQL server and shell.
- Cleans up to reduce image size.
- Creates a directory for initialization scripts.

6. MySQL data persistence

Dockerfile
VOLUME /var/lib/mysql
Show more lines

- Ensures MySQL data is stored outside the container for persistence.

7. Copy scripts

Dockerfile
COPY docker-entrypoint.sh /entrypoint.sh
COPY healthcheck.sh /healthcheck.sh
Show more lines

- Adds custom entrypoint and health check scripts into the container.

8. Entrypoint and Healthcheck

Dockerfile
ENTRYPOINT ["/entrypoint.sh"]
HEALTHCHECK CMD /healthcheck.sh
Show more lines

- ENTRYPOINT: Defines the main script that runs when the container starts (initializes MySQL).
- HEALTHCHECK: Docker will periodically run this script to check if MySQL is healthy.

9. Expose ports

Dockerfile
EXPOSE 3306 33060
Show more lines

- Opens MySQL ports:
  - 3306 → MySQL client connections.
  - 33060 → MySQL X Protocol.

10. Default command for MySQL

Dockerfile
CMD ["mysqld"]
Show more lines

- Runs the MySQL server process when the container starts.

☑ **What this Dockerfile does overall:**

- Combines **Nginx** and **MySQL** in one image (though this is not best practice; usually you separate services into different containers).
- Sets up volumes for persistence.
- Defines entrypoint and health checks for MySQL.
- Exposes necessary ports.

Let's clarify the difference between **CMD** and **ENTRYPOINT** in your example:

Your first Dockerfile

Dockerfile
FROM ubuntu
CMD ["sleep", "5"]
Show more lines

- Here, CMD sets the **default command** for the container: sleep 5.
- If you run:

Shell
docker run my-image
Show more lines
It executes sleep 5.

- If you override:

Shell
docker run my-image sleep 10
Show more lines
It replaces the entire CMD with sleep 10.

Alternative with ENTRYPOINT + CMD

Dockerfile
FROM ubuntu
ENTRYPOINT ["sleep"]
CMD ["5"]
Show more lines

- **ENTRYPOINT** defines the **base command** (sleep).
- **CMD** provides **default arguments** (5).
- When you run:

Shell
docker run my-image
Show more lines
It executes sleep 5.

- If you override:

Shell
docker run my-image 10
Show more lines
It becomes sleep 10 (because ENTRYPOINT stays, CMD changes).

☑ **Key Difference**

- **CMD only** → Entire command can be overridden.
- **ENTRYPOINT + CMD** → ENTRYPOINT is fixed, CMD is flexible for arguments.

Why use ENTRYPOINT + CMD?

- Useful when your container always runs the same program (e.g., sleep, nginx, python), but you want to allow changing arguments easily.

Summary of Docker Commands and Dockerfile Configurations

Below is a summary table of key commands and their effects:

Command/Configuration	Description	Example
Default CMD in Ubuntu container	Executes `bash` but exits when no terminal is attached	`docker run ubuntu`
Overridden CMD at runtime	Replaces default command with a specified one (e.g., `sleep 5`)	`docker run ubuntu sleep 5`
Permanent CMD update in Dockerfile	Builds an image with a permanent command change	`CMD ["sleep", "5"]`
Using ENTRYPOINT with CMD	Fixes the executable while allowing dynamic command-line argument replacement	`ENTRYPOINT ["sleep"]` & `CMD ["5"]`
Override ENTRYPOINT at runtime	Replaces the image's fixed executable with an alternative one	`docker run --entrypoint sleep2.0 ubuntu-sleeper 10`

Docker Compose

- Real-world apps often need **multiple services** (e.g., a web app + database + cache).
- Managing them individually with docker run is messy.
- Compose makes it **simple, repeatable, and organized**.

☑ What is Docker Compose?

Docker Compose is a tool that lets you **define and run multi-container applications** using a single configuration file (docker-compose.yml).
Instead of running multiple docker run commands manually, you describe everything in one file and start all services with **one command**.

☑ How it works

1. You create a docker-compose.yml file.
2. Define **services** (each service = one container).
3. Specify:
    - Image or build context
    - Ports
    - Volumes

- Environment variables

4. Start everything with:

Shell
docker-compose up

Docker-compose.yml

A Flask app with MySQL database:

```yaml
version: "3"
services:
  web:
    image: my-flask-app
    ports:
      - "5000:5000"
    depends_on:
      - db
  db:
    image: mysql:8
    environment:
      MYSQL_ROOT_PASSWORD: rootpass
      MYSQL_DATABASE: mydb
    volumes:
      - db_data:/var/lib/mysql

volumes:
  db_data:
```

What happens when you run docker-compose up?

- Starts **web** container (Flask app) and **db** container (MySQL).
- Maps ports, sets environment variables, and creates persistent volumes.

☑ Benefits

- **One command** to start/stop all services.
- **Easy scaling**: docker-compose up --scale web=3.
- **Portable**: Share docker-compose.yml with your team.

Yes, you **can change the name of docker-compose.yml**, but there's a small catch:

☑ Default behavior

- By default, Docker Compose looks for a file named:
  docker-compose.yml

  in the current directory.

☑ If you want a different name

- Use the -f (or --file) option:

Shell
docker-compose -f custom-name.yml up
Show more lines

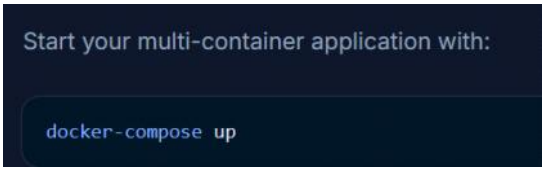
- You can even specify **multiple files**:

Shell
docker-compose -f base.yml -f override.yml up
Show more lines

☑ Why keep the default name?

- It's standard and makes collaboration easier.
- Tools and scripts often assume the default name.

In short: Yes, you can rename it, but you must explicitly tell Docker Compose using -f.

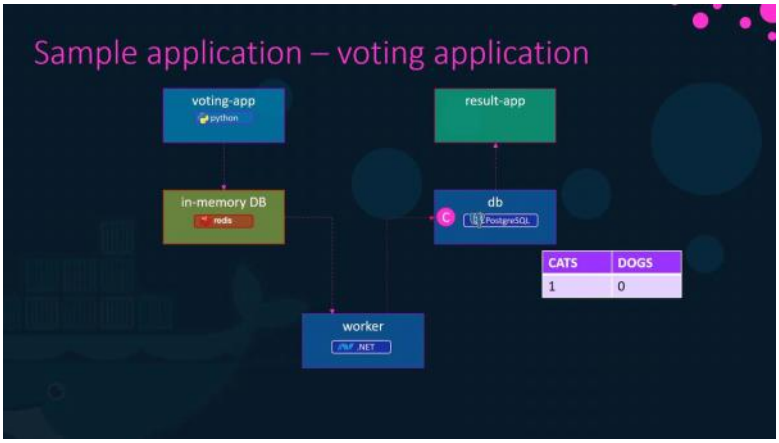


Sample Application: Voting App Architecture

To illustrate Docker Compose in practice, consider a sample voting application. This application demonstrates how Docker can integrate services built with different programming languages and frameworks.

Voting Application Components

- Python Front-End: A web interface that lets users vote between options (e.g., cat or dog). Votes are stored in a Redis instance, serving as an in-memory database.
- .NET Worker: A background service that processes votes and updates a PostgreSQL database with the vote counts.
- Node.js Back-End: A web application that displays voting results by reading data from PostgreSQL.



Deploying the Application Using Docker Run

To get started, let's deploy each layer of the voting application using individual Docker run commands. This example assumes that all necessary images are available in your Docker repository.

1. **Start a Redis Container:**

```
docker run -d --name=redis redis
```

2. **Deploy the PostgreSQL Database:**

```
docker run -d --name=db postgres
```

3. **Deploy the Voting Application (Front-End):**

   Map the container's port 80 to host port 5000.

```
docker run -d --name=vote -p 5000:80 voting-app
```

4. **Deploy the Results Web Application (Back-End):**

   Map the container's port 80 to host port 5001.

```
docker run -d --name=result -p 5001:80 result-app
```

5. **Deploy the Worker Container:**

```
docker run -d --name=worker worker
```

Linking Containers

To enable communication between containers, use the `--link` option. This option creates an entry in the container's `/etc/hosts` file for resolving the linked service by name.
For example, link the voting app container to Redis as follows:

```
docker run -d --name=vote -p 5000:80 --link redis:redis voting-app
```

Configuring Networks in Docker Compose

Docker Compose lets you define custom networks to control traffic between services

Creating the Docker Compose File

The next step is to set up your Docker Compose file for our application. This example deploys the following services:

- redis: the caching database
- db: the PostgreSQL database
- vote: the voting application
- worker: the background worker process
- result: the results viewer

Step 1: Create the File

Start by creating and writing to the Docker Compose file:

```
cat > docker-compose.yml
```

Step 2: Define the Service Configuration

Open the file for editing using your preferred text editor:

```
vi docker-compose.yml
```

Within this file, add your service definitions under the root level. A sample Docker Compose file is provided below:

```
redis:
  image: redis
db:
  image: postgres:9.4
vote:
  image: voting-app
  ports:
    - "5000:80"
worker:
  image: worker-app
result:
  image: result-app
  ports:
    - "5001:80"
```

In this configuration:

- The redis service uses the official Redis image.
- The db service uses PostgreSQL version 9.4.
- The vote service deploys the voting application, mapping port 5000 on the host to port 80 in the container.
- The worker service processes background tasks.
- The result service displays the outcome, mapping port 5001 on the host to port 80 in the container.

☑ What is Docker Engine?

Docker Engine is the **core component of Docker**.
It's the **runtime** that builds and runs containers. Think of it as the "brain" of Docker.

Main parts of Docker Engine

1. **Docker Daemon (dockerd)**
    - Runs in the background on your machine.

- Manages images, containers, networks, and volumes.
- Listens for API requests.

2. **Docker CLI (docker)**
   - The command-line tool you use (docker run, docker build, etc.).
   - Sends commands to the Docker Daemon via REST API.
3. **REST API**
   - The interface between CLI and Daemon.
   - Allows other tools (like Docker Compose) to talk to Docker.

☑ How it works

- You type:

```
Shell
docker run nginx
Show more lines
```

- CLI sends this request to the **Docker Daemon**.
- Daemon:
  - Pulls the image (if not present).
  - Creates a container.
  - Starts the container using the specified command.

What does "Containers isolate applications" mean?

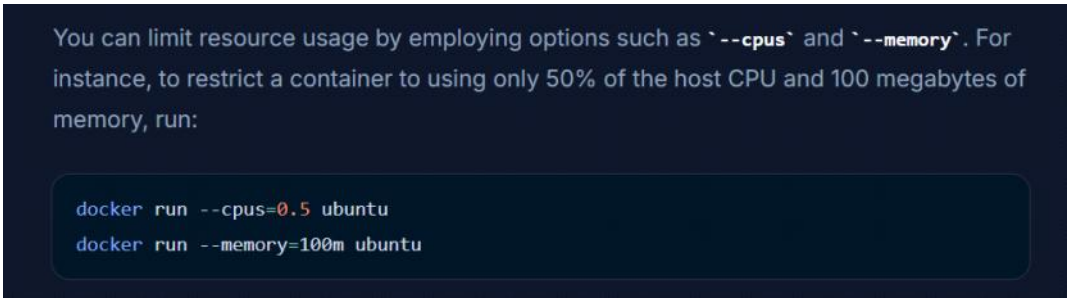
- Containers share the **same host OS kernel**, but they look like separate mini-systems.
- This isolation is achieved using **Linux namespaces**.
- Namespaces make each container think it has its own:
  - **Processes**
  - **Network**
  - **Filesystem**
  - **IPC (Inter-Process Communication)**
  - **Hostname**
- So, even though containers share hardware and OS, they **don't interfere with each other**.

☑ Understanding PID (Process ID) Namespace

- On Linux, when the system boots, the first process is **PID 1** (usually init or systemd).
- All other processes get unique PIDs globally.
- **Containers create their own PID namespace**, so:
  - Inside the container, the first process (like nginx) appears as **PID 1**.
  - On the host, the same process might be PID 3456.
- This gives the illusion that the container is an independent system.

Managing Resources with cgroups

Containers by default can use as much resource as they require, which may lead to resource exhaustion on the host. Docker utilizes Linux control groups (cgroups) to constrain the hardware resources available to each container, ensuring efficient resource management.



Learn where Docker stores its files, how it structures data, and how to handle persistent data effectively.

Docker Storage

When Docker is installed, it establishes a directory structure typically at /var/lib/docker. This root directory contains several subdirectories that serve different purposes:

- containers: Stores files related to running containers.
- images: Contains image-related files.
- volumes: Holds data for Docker volumes.
- overlay2: Manages the overlay filesystem for layering.

Command:

Shell
docker run -v data_volume:/var/lib/mysql mysql
Show more lines

☑ What does each part mean?

1. **docker run**
   Starts a new container.
2. **-v data_volume:/var/lib/mysql**
   This is **volume mapping**:
   - data_volume → A **named Docker volume** (created automatically if it doesn't exist).
   - /var/lib/mysql → The directory **inside the container** where MySQL stores its database files.
   - This means all MySQL data will be stored in data_volume on the host, so it **persists even if the container is removed**.
3. **mysql**
   The **image name** (official MySQL image from Docker Hub).

Container Orchestration

Container orchestration provides a robust solution to manage and scale containerized applications across multiple hosts.

With Docker, running a single application instance is straightforward.->
`docker run nodejs`

This command launches one instance of your application on a single Docker host.

you might be tempted to manually run additional instances by executing the Docker run command repeatedly:
`docker run nodejs`

For instance, if a container fails, you would need to manually redeploy it:
`docker run nodejs`

Moreover, a failure of the Docker host itself renders all containers on that host inaccessible.

Container orchestration is a comprehensive set of tools and procedures designed to automate the deployment, scaling, and management of containerized applications. An effective orchestration solution spans multiple Docker hosts, ensuring that if one host fails, the application remains available through other hosts.

ensuring that if one host fails, the application remains available through other hosts.

For instance, using Docker Swarm you can scale your Node.js application by simply running:

```
docker service create --replicas=100 nodejs
```

This command deploys a service with 100 replicas, dramatically simplifying the scaling process.

Benefits of Container Orchestration

Container orchestration solutions offer several advanced features that streamline production deployments:

- **Automatic Scaling:** Dynamically adjust the number of container instances based on load.
- **High Availability:** Distribute containers across multiple hosts, ensuring continued service even if one host fails.
- **Advanced Networking:** Enable seamless communication between containers across hosts and implement load balancing for incoming requests.
- **Centralized Storage Management:** Provide persistent data sharing alongside centralized configuration and security management.

Popular Container Orchestration Tools

Several orchestration platforms have emerged to meet the demands of different environments:

Platform	Key Features	Use Case
Docker Swarm	Simplified setup & management	Quick deployments with moderate scaling
Kubernetes	Extensive customization & multi-cloud support	Complex, large-scale production environments
Apache Mesos	Advanced features with higher configuration complexity	Highly specialized or custom orchestrations

What is Docker Swarm?

Docker Swarm allows you to combine multiple Docker hosts into a single cluster.

Within the swarm cluster, the manager node orchestrates the distribution of your services (or application instances) across different hosts, ensuring high availability and effective load balancing.

To set up Docker Swarm, ensure you have several hosts with Docker installed.

Designate one host as the manager (sometimes known as the master or swarm manager) and the others as worker nodes.

Initializing the Swarm

Once your hosts are ready, begin by initializing the Docker Swarm on the manager node using the following command:

```
docker swarm init --advertise-addr <manager-ip>
```

When you run this command, Docker Swarm outputs a join command for the worker nodes. For example, the output might look like this:

```
root@osboxes:/root/simple-webapp-docker # docker swarm init --advertise-addr 192.168.1
Swarm initialized: current node (0j76dum2r56p1xfn4u1pls2c) is now a manager.
To add a worker to this swarm, run the following command:
    docker swarm join --token SWMTKN-1-35va8b3fi5krpdkefqxgtmulw3z828daucri7y526ne0sgu
To add a manager to this swarm, run 'docker swarm join-token manager' and follow the i
```

Deploying Services Using Docker Swarm

Traditionally, you might deploy an application instance (such as a web server) using the `docker run` command. However, running `docker run` individually on each worker node is impractical for large clusters. It requires manual intervention for deployment, load balancing, and monitoring.

Docker Swarm addresses these challenges by orchestrating containers through Docker services. Docker services let you run one or more instances (replicas) of an application across your cluster nodes.

For example, to deploy multiple instances (replicas) of your web server application across worker nodes, execute the following command on the manager node:

```
docker service create --replicas=3 -p 8080:80 --network frontend my-web-server
```

Beyond simple scaling, Kubernetes supports dynamic adjustments—up to two thousand instances with another command. It also facilitates rolling upgrades, allowing you to update one instance at a time, and includes a rollback mechanism if issues arise.

Kubernetes Architecture Overview

kubectl is the essential CLI tool for deploying and managing applications on a Kubernetes cluster.Kubernetes leverages container runtimes like Docker, Rocket, or CRI-O to execute applications within isolated containers. A Kubernetes cluster consists of multiple nodes managed by a master node that orchestrates containerized applications.

Nodes

Nodes are physical or virtual machines running Kubernetes software. They act as workers where containers are deployed. A typical Kubernetes cluster includes multiple nodes to guarantee high availability; if one node fails, the remaining nodes continue to operate, ensuring uninterrupted service.

Master

The master node runs the control plane components that manage and monitor the state of the cluster. Its responsibilities include maintaining cluster health, monitoring nodes, and reallocating workloads in the event of node failures.

When installing Kubernetes, the following core components are set up:

- API Server: Serves as the primary interface for all administrative tasks. It allows users, management devices, and CLI tools to communicate with the Kubernetes cluster.
- etcd: A distributed and reliable key-value store that contains all cluster data, ensuring consistency across multiple nodes.
- Scheduler: Responsible for distributing newly created containers across available nodes.
- Controllers: Detect changes in the cluster (like node failures or container issues) and initiate corrective actions by deploying new containers as needed.
- Container Runtime: The underlying software (e.g., Docker) that runs containers.
- kubelet: An agent that runs on each node to ensure containers operate as expected.

The diagram below illustrates these components and their interactions:

kubectl is the essential CLI tool for deploying and managing applications on a Kubernetes cluster.

Kubernetes for Absolute Beginners

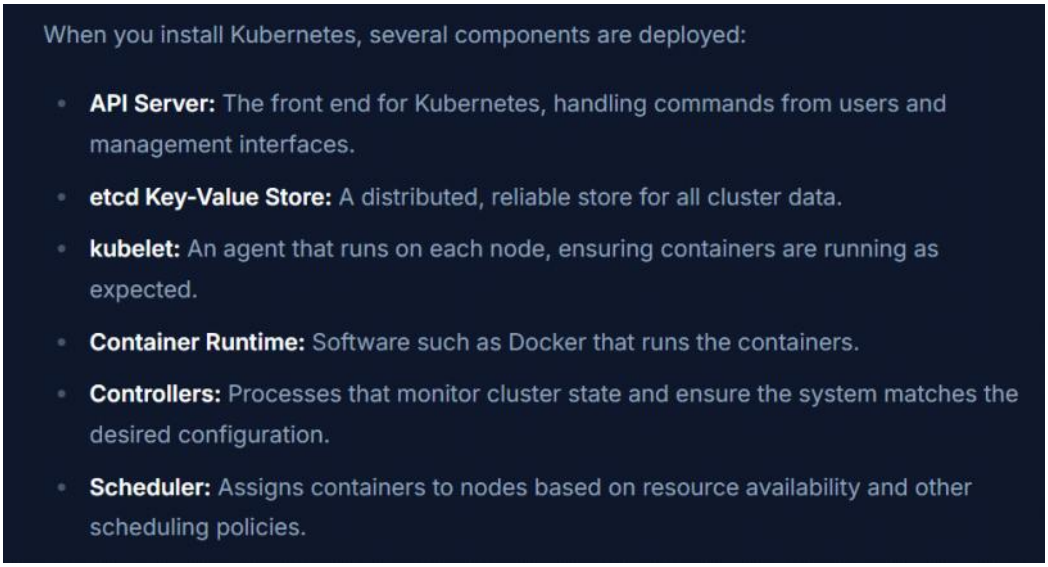
Kubernetes can manage and scale thousands of containers across a clustered environment.

Node

A node is a physical or virtual machine where Kubernetes is installed. -> acting as worker in your cluster.

Nodes run our containerized app.

The master node hosts components that control and monitor the cluster's state.



Here's a **simplified but still technical explanation** of each Kubernetes component's role:

1. API Server

- **Role:** Acts as the central communication hub.
- **Easy way:** It's the **entry point** for all commands and queries. Every request from kubectl or other tools goes through the API Server.

2. etcd

- **Role:** Stores the entire cluster state in a consistent, distributed key-value database.
- **Easy way:** Think of it as the **source of truth** for Kubernetes—every configuration and status is recorded here.

3. kubelet

- **Role:** Runs on each node and ensures containers are healthy and match the desired state.
- **Easy way:** It's the **node-level supervisor** that talks to the API Server and manages containers locally.

4. Container Runtime

- **Role:** Executes containers based on images.
- **Easy way:** It's the **engine** that actually runs your application inside containers (e.g., Docker, containerd).

5. Controllers

- **Role:** Continuously monitor cluster state and reconcile it with the desired configuration.
- **Easy way:** They are the **automated fixers**—if something drifts from what you declared, controllers correct it.

6. Scheduler

- **Role:** Decides which node will run a new Pod based on resource availability and constraints.
- **Easy way:** It's the **planner** that picks the best spot for your workload.

Would you like me to **put these into a quick mnemonic or cheat sheet** so you can recall them instantly? Or maybe a **one-line summary for each in a table format**?

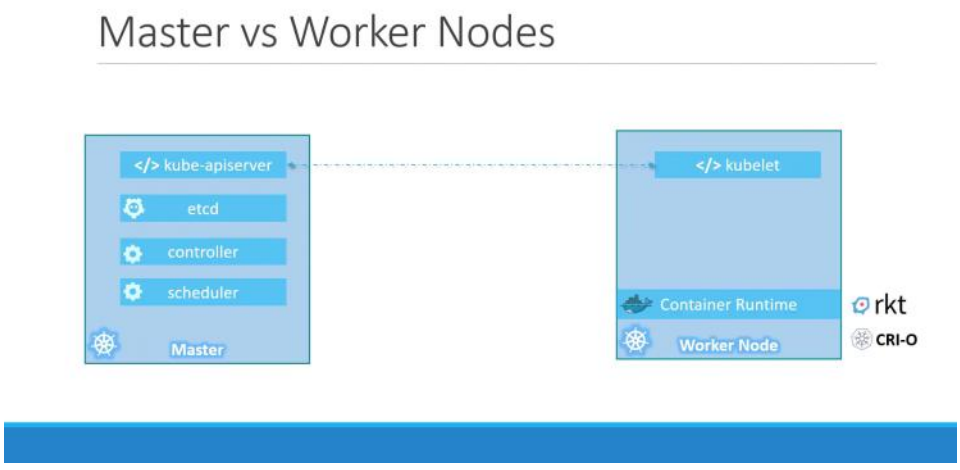
etcd is a reliable distributed key-value store used for saving all cluster data

The scheduler is responsible for detecting newly created container requests and assigning them to the most suitable nodes based on available resources and defined policies.

Controllers -> making decisions such as restarting containers when failures occur.

Container runtime -> running containers (docker,crio)

Kubelet -> runs on every node -> ensure each container is healthy -> communicates with master nodes about node status



With kubectl, you can retrieve detailed cluster information, monitor node status, and manage workloads.

Containerd originated as a component of Docker but has evolved into an independent project under the Cloud Native Computing Foundation with graduated status

Kubernetes **CRI (Container Runtime Interface) tools** are basically the bridge between Kubernetes and the container runtime. Here's a **lucid, technical breakdown**:

What is CRI?

- CRI = **Container Runtime Interface**, a standard API that Kubernetes uses to talk to container

runtimes.
- It allows Kubernetes to work with different runtimes without hardcoding Docker or any specific runtime.

Key CRI Tools

1. **containerd**
   - A lightweight, industry-standard container runtime.
   - Handles pulling images, managing containers, and storage.
   - **Why popular?** It's simple, fast, and directly supported by Kubernetes.
2. **CRI-O**
   - Purpose-built for Kubernetes.
   - Implements CRI using **Open Container Initiative (OCI)** standards.
   - **Why use it?** Minimal overhead, secure, and optimized for Kubernetes.
3. **Docker (via dockershim)** *(Deprecated)*
   - Kubernetes used to talk to Docker through dockershim.
   - Removed in Kubernetes v1.24 because containerd and CRI-O are now preferred.

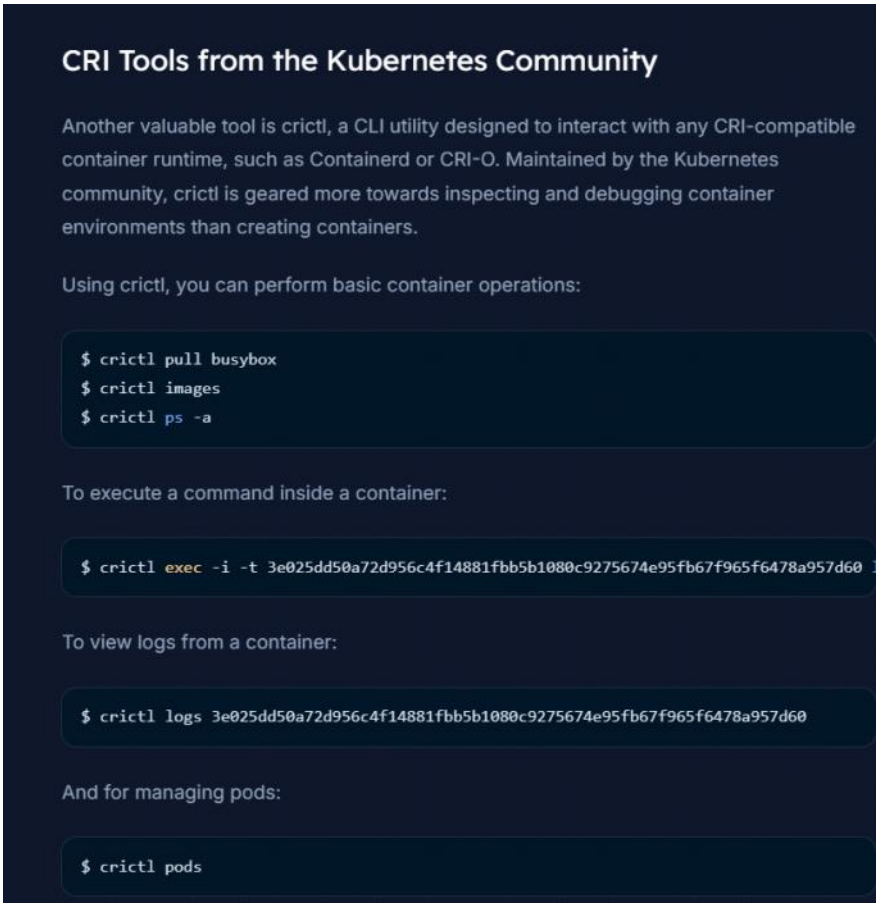
How They Work

- kubelet uses CRI to send requests like:
  - **CreateContainer**
  - **StartContainer**
  - **PullImage**
- The runtime (containerd or CRI-O) executes these actions.

☑ **Easy way to remember:**

- **CRI = Common Language** between Kubernetes and runtimes.
- **containerd & CRI-O = Translators** that speak this language.

Do you want me to **make a quick comparison table of containerd vs CRI-O vs Docker** for clarity? Or a **diagram showing how kubelet talks to CRI tools**?	



Here's a **lucid technical comparison** of **CRI-O vs containerd vs Docker**:

☑ 1. Docker

- **What it is:** A full container platform (runtime + build + image management).
- **Role in Kubernetes:** Used to be the default runtime, but **deprecated since v1.24** because Kubernetes now uses CRI-compliant runtimes.
- **Pros:** Mature ecosystem, great for development.
- **Cons:** Extra layer (dockershim) needed for Kubernetes → more complexity.

☑ 2. containerd

- **What it is:** A lightweight container runtime originally part of Docker, now standalone.
- **Role in Kubernetes:** Implements **CRI**, so kubelet talks directly to containerd.
- **Pros:** Simple, fast, widely adopted, CNCF project.
- **Cons:** No built-in image build tools (needs external tools like buildkit).

☑ 3. CRI-O

- **What it is:** A Kubernetes-specific container runtime.
- **Role in Kubernetes:** Implements **CRI** using OCI standards.
- **Pros:** Minimal, secure, optimized for Kubernetes only.
- **Cons:** Less general-purpose than containerd (not for non-K8s workloads).

Quick Summary Table

Feature	Docker	containerd	CRI-O
CRI Support	✗ (via dockershim, now removed)	☑	☑
Focus	Full container platform	General-purpose runtime	Kubernetes-only runtime

Extra Layer	Yes (dockershim)	No	No
Image Build	Built-in	External (BuildKit)	External
Performance	Moderate	High	High
Current Status	Deprecated for K8s	Recommended	Recommended

☑ **Easy way to remember:**

- **Docker** = Old default, now deprecated for K8s.
- **containerd** = Lightweight, general-purpose, widely used.
- **CRI-O** = Kubernetes-specialist runtime.

Do you want me to **create a visual diagram showing how kubelet interacts with these runtimes**?
Or a **flowchart of Kubernetes → CRI → Runtime**?

☑ What is an Endpoint in Kubernetes?

- An **Endpoints object** is created automatically when you create a **Service**.
- It contains the **IP addresses and ports of the Pods** that match the Service's selector.
- The Service uses this list to **forward traffic to the right Pods**.

Why is it needed?

- Pods are dynamic (IP changes when they restart).
- Services give a **stable virtual IP**, but they need to know **where the actual Pods are**.
- Endpoints solve this by maintaining the mapping.

Flow

Client → Service (ClusterIP) → Endpoints → Pods

Example

If you create a Service named my-service for Pods with label app=nginx, Kubernetes will:

- Find all Pods with app=nginx.
- Create an Endpoints object like:

YAML

```
apiVersion: v1
kind: Endpoints
metadata:
 name: my-service
subsets:
 - addresses:
 - ip: 10.244.1.5
 - ip: 10.244.2.7
 ports:
 - port: 80
```

☑ **Easy way to remember:**

- **Service = Virtual IP**
- **Endpoints = Real Pod IPs behind the Service**

When we say **Pods are ephemeral**, it means:

- **Pods are temporary and short-lived** in nature.
- They **don't have a fixed identity or IP address**—if a Pod crashes or is deleted, Kubernetes can create a new Pod with a different IP.
- They are **not designed to store permanent data** because they can be replaced anytime by the scheduler or controllers.
- Their lifecycle is tied to workloads: when scaling up/down or updating, old Pods are terminated and new ones are created.

☑ **Key takeaway:**
Pods are **disposable units**—Kubernetes treats them as replaceable building blocks. If you need persistence, you use **Persistent Volumes** or higher-level abstractions like **Deployments** to manage Pod replacement.

Originally, Docker was the sole container runtime supported by Kubernetes. However, to accommodate additional runtimes, Kubernetes introduced the Container Runtime Interface (CRI).

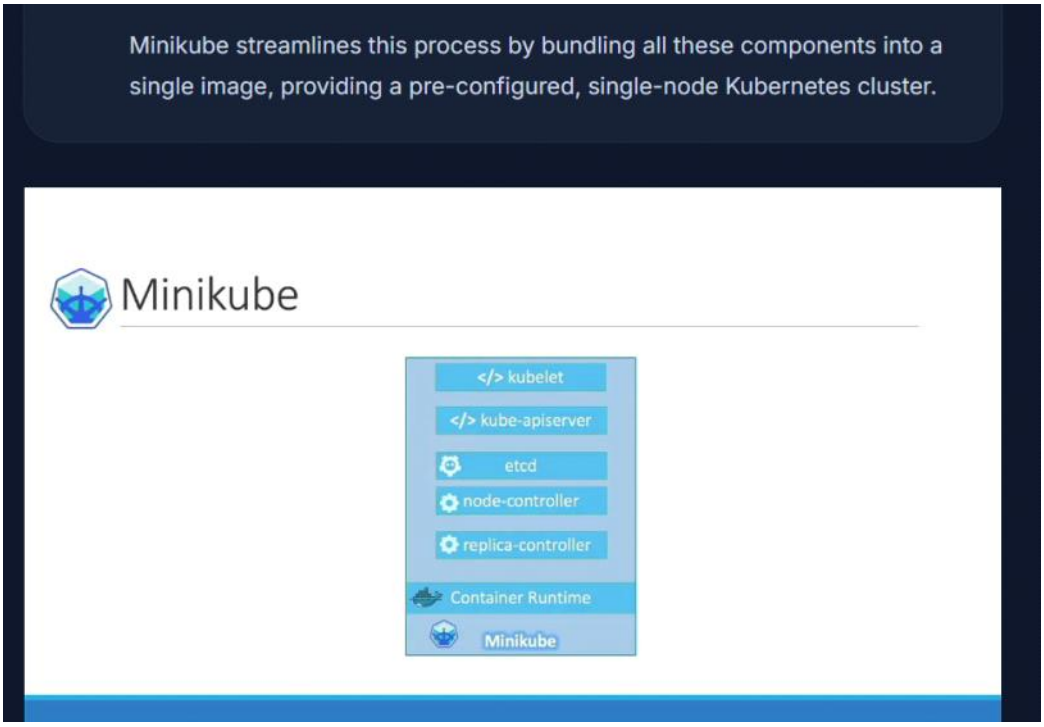
Docker also featured a daemon that managed the runtime called containerd.

Containerd is fully CRI-compatible and can interact directly with Kubernetes as other runtimes do

This means containerd functions as a standalone runtime independent of Docker, allowing Kubernetes to manage functionalities like the Docker CLI, API, and build tools on its own.

Although Kubernetes has deprecated Docker as its runtime, Docker itself remains an integral tool in container management.Even if Docker is not installed on your machine, or if you are exclusively using containerd

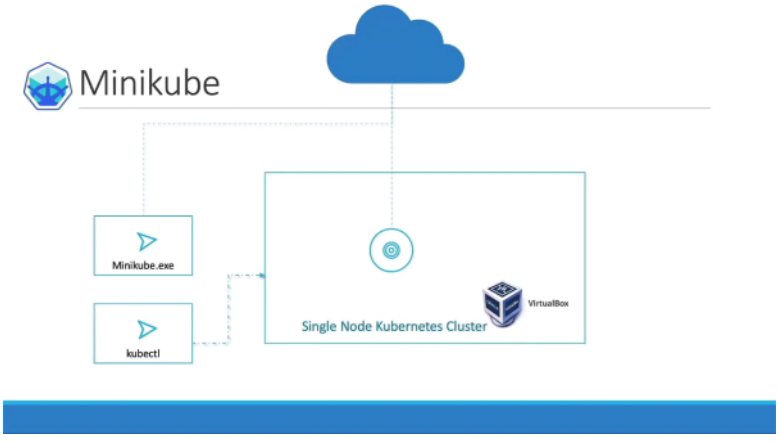
Minikube is the easiest way to launch a local Kubernetes cluster.



Ensure that a compatible hypervisor (e.g., VirtualBox, Hyper-V, or KVM) is installed on your system before running Minikube.

Minikube packages the complete Kubernetes bundle into an ISO image that is automatically downloaded and deployed using its command-line utility. It integrates seamlessly with various virtualization platforms, such as:

- Oracle VirtualBox
- VMware Fusion
- Hyper-V (for Windows users)
- KVM (for Linux users)



Some assumptions before we start off with pods:

• Your application has already been developed and packaged as Docker images.
• These Docker images are stored in a repository, such as Docker Hub, so Kubernetes can pull them as needed.
• Your Kubernetes cluster is set up and running, whether on a single node or across multiple nodes, with all services operational.

Pod:
Kubernetes deploys your application by encapsulating containers inside a fundamental unit called a pod.

A pod represents a single instance of your application and is the smallest deployable object in Kubernetes.

Scaling

As your user base increases, scaling becomes necessary.

Instead of increasing the number of containers within the same pod, you create additional pods—each running an instance of your application.

If your current node cannot handle the load, Kubernetes can schedule more pods on new nodes

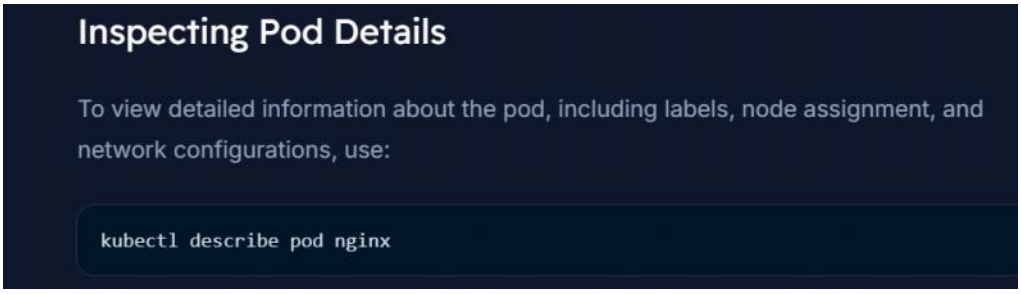
a pod can host multiple containers.

Deploying a Pod with kubectl

kubectl run nginx --image nginx -> This command creates a pod running an instance of the nginx Docker image, pulling the image from Docker Hub or another configured repository

So, now our pod running nginx server is created

kubectl get pods -> we can view this pod.



We can also create a a Kubernetes Pod using a YAML configuration file.

Kubernetes resource definition files follow a consistent structure and always include four top-level fields:

- apiVersion
- kind
- metadata
- spec

Below is an example of a complete YAML configuration for a Pod:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
    type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx
```

Creating the Pod

To create the Pod from your YAML file, run the following command:

```
kubectl create -f pod-definition.yml
```

Verifying the Pod

Once the Pod is created, you can verify its status using several `kubectl` commands.

Listing All Pods

To list all Pods in your cluster, use:

```
kubectl get pods
```

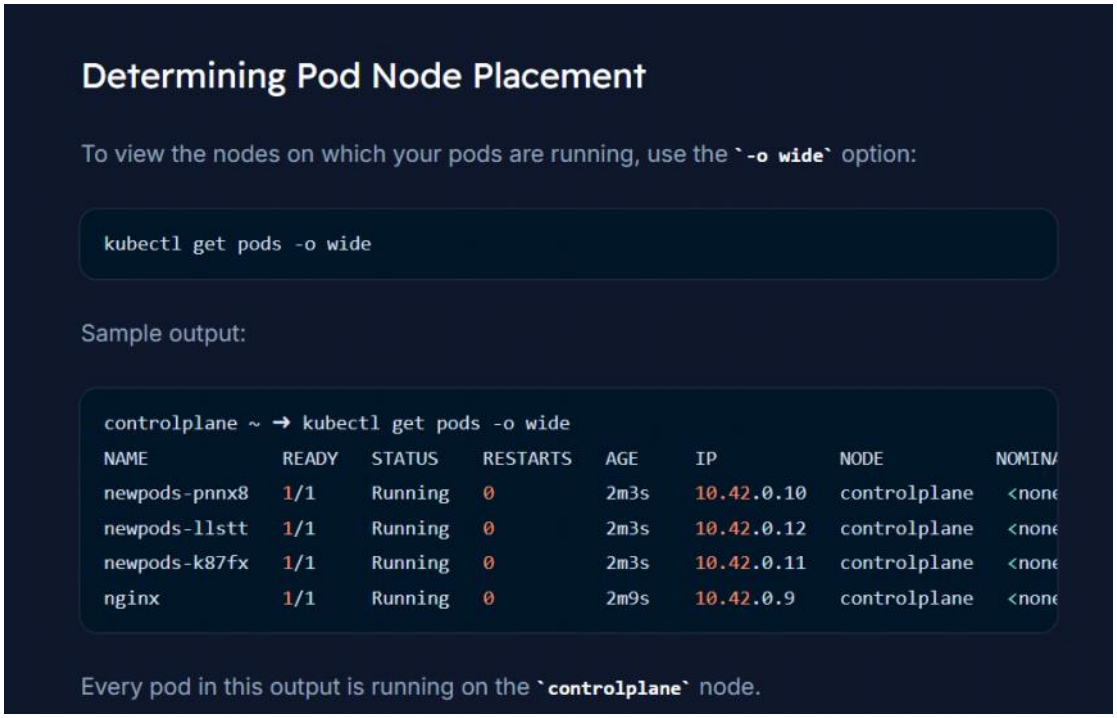
Example output:

```
NAME       READY   STATUS    RESTARTS   AGE
myapp-pod  1/1     Running   0          20s
```

Viewing Detailed Pod Information

For detailed information about the Pod, execute:

```
kubectl describe pod myapp-pod
```



Deleting the Web App Pod

If you need to remove the webapp pod, execute the following command:

```
kubectl delete pod webapp
```

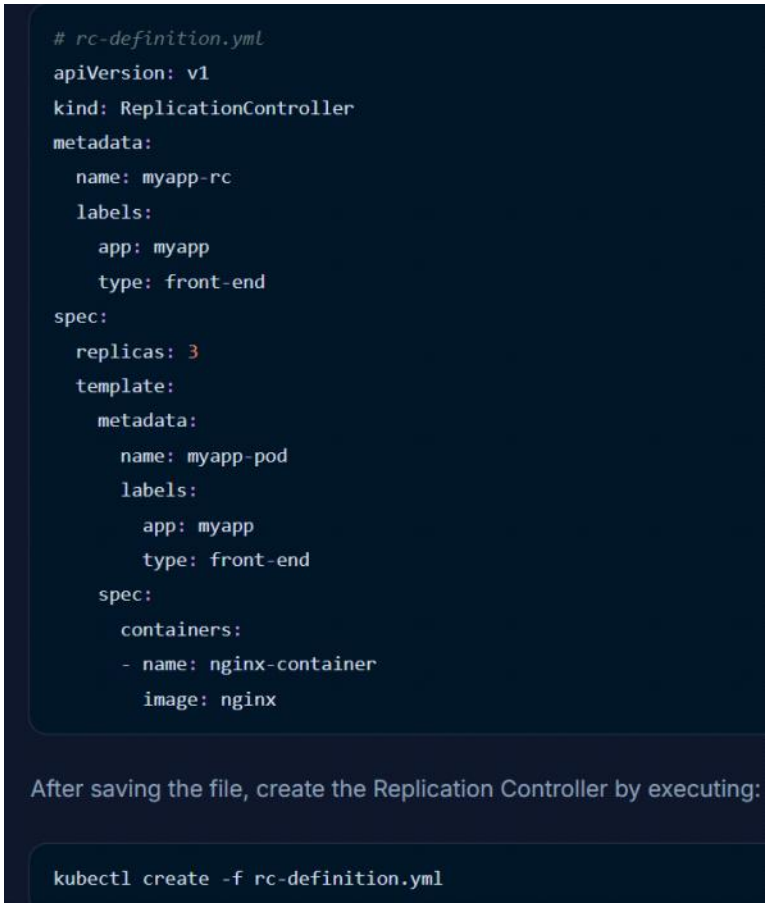
Imagine an application running on a single pod. If that pod fails, your application becomes unavailable. The Replication Controller prevents this by maintaining multiple instances of a pod.

Even if you choose to run only one pod, the Replication Controller immediately replaces a failed pod, guaranteeing that the desired number of pods remains active.

When user demand increases, additional pods can be deployed, balancing the load across nodes and enhancing performance.

The ReplicaSet replaces the older Replication Controller. ReplicaSets offer a more explicit configuration, particularly with required label selectors.

Less create a replication controller



kubectl get replicationcontroller

To view the individual pods managed by the replication controller, run:
kubectl get pods
Pods created by the controller will typically begin with the name "myapp-rc," indicating they are managed automatically.

ReplicaSets are similar to Replication Controllers but use the apps/v1 API version and require an explicit selector. The selector field determines which pods the ReplicaSet should manage by matching labels.

Below is an example of a ReplicaSet definition file named **`replicaset-definition.yml`**:

```yaml
# replicaset-definition.yml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
    app: myapp
    type: front-end
spec:
  replicas: 3
  selector:
    matchLabels:
      type: front-end
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx
```

To create the ReplicaSet, run:

```
kubectl create -f replicaset-definition.yml
```

Verify the creation of your ReplicaSet with:

```
kubectl get replicaset
```

And check the pods by executing:

```
kubectl get pods
```

Scaling a ReplicaSet

Scaling a ReplicaSet allows you to adjust the number of pod replicas based on demand. Suppose you start with three replicas and later need to scale to six. You have two options:

1. Update the `replicas` number in your definition file (change from `3` to `6`) and apply the change:

```
kubectl replace -f replicaset-definition.yml
```

2. Use the `kubectl scale` command directly. You can specify the new replica count using either the file or the ReplicaSet name:

```
kubectl scale --replicas=6 -f replicaset-definition.yml
```

or

```
kubectl scale --replicas=6 replicaset myapp-replicaset
```

Essential Kubernetes Commands

Below is a summary of common commands to manage Replication Controllers and ReplicaSets:

Operation	Command	Example
Create objects	kubectl create	`kubectl create -f rc-definition.yml`
View objects	kubectl get	`kubectl get replicaset` `kubectl get pods`
Delete ReplicaSet	kubectl delete	`kubectl delete replicaset myapp-replicaset`
Update configuration	kubectl replace	`kubectl replace -f replicaset-definition.yml`
Scale ReplicaSet	kubectl scale	`kubectl scale --replicas=6 replicaset myapp-replicaset`

Rolling Update and Deployment

You require multiple instances of this web server to handle load distribution and high availability

Moreover, when a new version of the application is available on the Docker registry, you want to upgrade your instances one by one -> **rolling update**

In case an update introduces an error, a quick **rollback** mechanism is essential.

While replicasets manage pods ensuring correct no. of them are running, Deployments not only handles rolling updates and rollbacks but also lets you pause and resume deployments as needed.

The Deployment sits at the top of this hierarchy, enabling advanced management features such as rolling updates, rollbacks, and dynamic pausing/resuming of deployments.

To create a Deployment, you first define a deployment configuration file. This file's structure is similar to that of a ReplicaSet, with the key difference being that the `kind` is set to "Deployment". Below is an example of a Deployment manifest:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
    app: myapp
    type: front-end
spec:
  replicas: 3
  selector:
    matchLabels:
      type: front-end
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx
```

In this configuration:

- The `apiVersion` is set to `apps/v1`.

Verifying the Deployment

After creation, verify the Deployment, ReplicaSet, and Pods using these commands:

1. **Check Deployments:**

```
kubectl get deployments
```

Example output:

```
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
myapp-deployment   3         3         3            3           21s
```

2. **View the ReplicaSet:**

```
kubectl get replicaset
```

Example output:

```
NAME                          DESIRED   CURRENT   READY   AGE
myapp-deployment-6795844b58   3         3         3       2m
```

3. **Examine the Pods:**

```
kubectl get pods
```

Viewing All Created Objects

To see all Kubernetes objects associated with your Deployment at once, run:

```
kubectl get all
```

Example output:

```
NAME                                   DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   A
deploy/myapp-deployment                3         3         3            3           9
NAME                                   DESIRED   CURRENT   READY   AGE
rs/myapp-deployment-6795844b58         3         3         3       9h
NAME                                   READY   STATUS    RESTARTS   AGE
po/myapp-deployment-6795844b58-5rbjl   1/1     Running   0          9h
po/myapp-deployment-6795844b58-h4w55   1/1     Running   0          9h
po/myapp-deployment-6795844b58-1fjhv   1/1     Running   0          9h
```

To check the status of a rollout, execute the following command:

```
> kubectl rollout status deployment/myapp-deployment
Waiting for rollout to finish: 0 of 10 updated replicas are available...
Waiting for rollout to finish: 1 of 10 updated replicas are available...
Waiting for rollout to finish: 2 of 10 updated replicas are available...
Waiting for rollout to finish: 3 of 10 updated replicas are available...
Waiting for rollout to finish: 4 of 10 updated replicas are available...
Waiting for rollout to finish: 5 of 10 updated replicas are available...
Waiting for rollout to finish: 6 of 10 updated replicas are available...
Waiting for rollout to finish: 7 of 10 updated replicas are available...
Waiting for rollout to finish: 8 of 10 updated replicas are available...
Waiting for rollout to finish: 9 of 10 updated replicas are available...
deployment "myapp-deployment" successfully rolled out
```

You can also view the rollout history with:

```
> kubectl rollout history deployment/myapp-deployment
```

Deployment Strategies

There are two primary deployment strategies in Kubernetes:

1. Recreate Strategy:
   In this method, when you have multiple replicas (such as five instances of your application), all existing instances are terminated before the new instances are deployed. This approach, while straightforward, causes downtime between shutting down the old pods and starting the new ones.
2. Rolling Update Strategy:
   Here, the new version gradually replaces the old version without impacting application availability. Kubernetes incrementally scales up the new pods while scaling down the old ones.

If no strategy is specified, Kubernetes defaults to the rolling update strategy.

Updating a Deployment

1. Update your deployment YAML file (for example, deployment-definition.yml) by modifying

parameters like the container image version.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
    app: myapp
    type: front-end
spec:
  replicas: 3
  selector:
    matchLabels:
      type: front-end
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.7.1
```

After saving the file, apply the changes using:

```
kubectl apply -f deployment-definition.yml
```

This command triggers a new rollout and generates a new deployment revision.

2. If you want to quickly update the container image without changing the rest of the configuration, use:

```
kubectl set image deployment/myapp-deployment nginx-container=nginx:1.9.1
```

When using the recreate strategy, you may see entries showing that the old ReplicaSet is scaled down to zero before the new ReplicaSet is scaled up.

For the rolling update strategy, the output indicates a gradual scaling of the old and new ReplicaSets

When a deployment is created with multiple replicas (for example, five), Kubernetes automatically generates a ReplicaSet that manages pod creation

During an upgrade, a new ReplicaSet is created with the updated configuration while the old ReplicaSet gradually scales down.

Rolling Back a Deployment

If issues are detected with the new version after an upgrade, Kubernetes makes it simple to roll back to a previous working deployment. Execute the following command to undo the latest rollout:

```
kubectl rollout undo deployment/myapp-deployment
```

This command stops the new ReplicaSet, scales down its pods, and scales up the pods from the previous ReplicaSet.

Before the rollback, the output may appear similar to:

```
> kubectl get replicasets
NAME                          DESIRED   CURRENT   READY   AGE
myapp-deployment-67c749c58c   0         0         0       22m
myapp-deployment-7d57dbd8d    5         5         5       20m
```

After executing the rollback, the ReplicaSets will reverse their roles:

```
> kubectl get replicasets
NAME                          DESIRED   CURRENT   READY   AGE
myapp-deployment-67c749c58c   5         5         5       22m
myapp-deployment-7d57dbd8d    0         0         0       20m
```

Verify the rollback by comparing the output of kubectl get replicasets before and after the command.

Summary of Essential Commands

Below is a table summarizing key commands for managing deployments in Kubernetes:

Command	Description
`kubectl create -f deployment-definition.yml`	Create a new deployment from the YAML definition
`kubectl get deployments`	List all deployments
`kubectl apply -f deployment-definition.yml`	Apply updates to the deployment from the YAML file
`kubectl set image deployment/myapp-deployment nginx-container=nginx:1.9.1`	Update the image for a specific container in the deployment
`kubectl rollout status deployment/myapp-deployment`	Check the status of the rollout
`kubectl rollout undo deployment/myapp-deployment`	Roll back to the previous deployment revision

Let me break this down in a simple way:

☑ Single-Node Kubernetes Cluster

- You have **one machine (node)** running Kubernetes.
- This node has an IP like 192.168.1.2 (used for SSH or management).
- Inside this node, Kubernetes creates an **internal network** (e.g., 10.244.0.0/16) for **pods**.
- Each **pod** gets its own IP from this internal range (e.g., 10.244.0.2).
- These pod IPs are **not the same as your node IP** and **not stable** (they change if the pod restarts).
- Pods can talk to each other directly using these internal IPs.

☑ Multi-Node Kubernetes Cluster

- Now you have **two nodes**, say:
  - Node 1: 192.168.1.2
  - Node 2: 192.168.1.3
- Each node runs pods and uses the same internal range (10.244.0.0/16).
- If both nodes assign 10.244.0.2 to different pods, **IP conflict happens**.
- Kubernetes **does not fix this automatically**.
- To avoid conflicts, you need a **CNI (Container Network Interface)** plugin like **Flannel, Calico, Weave**, which ensures **unique pod IPs across all nodes**.

☑ Networking Principles

1. **Pods must talk to each other without NAT** (direct communication).
2. **Nodes must talk to pods and vice versa**.
3. This is why Kubernetes uses **overlay networks** via CNI plugins.

🔑 Key Idea

- Node IPs (e.g., 192.168.x.x) are for **machines**.
- Pod IPs (e.g., 10.244.x.x) are for **containers inside Kubernetes**.
- In multi-node clusters, you need a networking solution so pod IPs are **unique across the**

cluster.

Would you like me to **draw a simple diagram** showing:

- Node IPs vs Pod IPs
- How networking works in single-node vs multi-node clusters?
  Or should I **explain how CNI plugins solve this problem**?

Here's the concept in **plain language**:

☑ Why Do We Need a Networking Solution?

- Kubernetes **does not handle pod networking by itself**.
- In a cluster, every pod needs a **unique IP address** so they can talk to each other and to nodes.
- If you have multiple nodes, you can't just use the same IP range on each node (it would cause conflicts).
- So, Kubernetes uses **external networking solutions** (called **CNI plugins**) to manage this.

☑ What Do These Solutions Do?

- They **divide the cluster network into segments**.
- Each node gets its **own subnet** (a slice of the overall IP range).
- Pods on that node get IPs from that subnet.
- This way, **no two pods in the cluster share the same IP**.
- They also set up **routing rules** so:
  - Pods on Node A can talk to pods on Node B.
  - Nodes can talk to pods, and vice versa.

☑ Examples of Solutions

- **On-Premises**: Calico, Flannel (simple and widely used).
- **VMware setups**: NSX-T (integrates with VMware networking).
- **Lab/testing**: Weave Net (easy to set up for experiments).

☑ How It Works (Simplified)

Imagine:

- Node 1 gets subnet 10.244.1.0/24
- Node 2 gets subnet 10.244.2.0/24
- Pod on Node 1 → IP 10.244.1.5
- Pod on Node 2 → IP 10.244.2.8
- The CNI plugin ensures **routing between these subnets**, so pods can talk directly without NAT.

☑ Why Is This Important?

- Without this, pods on different nodes **couldn't communicate reliably**.
- Networking solutions make the cluster **scalable and conflict-free**.

Would you like me to:

1. **Draw a simple diagram** showing nodes, subnets, and pod IPs?
2. Or **give you a step-by-step guide** on how Flannel or Calico actually sets this up?
3. Or **compare these solutions** (Calico vs Flannel vs Weave) in terms of performance and features?

☑ Why “without NAT”?

- NAT (Network Address Translation) changes IP addresses during communication.
- Kubernetes networking model says:
  - **Pod IPs should be visible and reachable directly across the cluster**.
- So, instead of hiding pod IPs behind node IPs (like NAT would), the plugin ensures:
  - Pod A (10.244.1.5) can directly talk to Pod B (10.244.2.8) using their real IPs.

Let’s make this **super simple**:

☑ Why Do We Need Services in Kubernetes?

- Pods have **dynamic IPs** (they change when pods restart).
- If your app has multiple pods (frontend, backend), hardcoding IPs is impossible.
- **Services** give a **stable way** to access pods, even if their IPs change.

☑ Problem with External Access

- Pod IPs (like 10.244.0.2) are **internal** to the cluster.
- Your laptop (e.g., 192.168.1.10) **cannot directly reach** that IP.
- You want to access the app from outside without SSH into the node.

☑ Solution: NodePort Service

- A **NodePort Service** opens a port on **every node** in the cluster.
- Example:
  - Node IP: 192.168.1.2
  - NodePort: 30008 (range: 30000–32767)
  - Pod Port: 80 (web server)
- When you hit http://192.168.1.2:30008, Kubernetes forwards traffic to the pod’s port 80.

☑ How It Works

- **Three ports involved**:
  1. **TargetPort** → Pod’s port (e.g., 80)
  2. **Port** → Service’s internal port (e.g., 80)
  3. **NodePort** → External port on node (e.g., 30008)
- Kubernetes maps: Node IP:NodePort → Service → Pod IP:TargetPort

☑ YAML Example

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  type: NodePort
  ports:
    - targetPort: 80
      port: 80
      nodePort: 30008
  selector:
    app: myapp
    type: front-end
```

- selector matches pods with labels app: myapp and type: front-end.

☑ Access

- From outside:\ http://<Node-IP>:<NodePort> → e.g., http://192.168.1.2:30008

☑ Load Balancing

- If multiple pods have the same label, the service **distributes traffic** among them.
- Works across **multiple nodes** too. Any node's IP + NodePort will reach the service.

☑ Other Service Types

- **ClusterIP** → Internal-only communication (default).
- **LoadBalancer** → Cloud environments (creates external load balancer).

In short:\ NodePort = external access via node's IP and a fixed port.\ ClusterIP = internal access only.
\ LoadBalancer = external access with cloud load balancer.

👉 Do you want me to **draw a simple diagram** showing:

- NodePort mapping
- How traffic flows from user → node → pod?\ Or should I **give you a quick analogy** (like a receptionist routing calls) to make it even easier?

When Pods share the same labels, Kubernetes automatically registers all matching Pods as endpoints for the Service.

For clarity, here is a corresponding Pod definition:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
    type: front-end
spec:
  containers:
    - name: nginx-container
      image: nginx
```

For example, if there are three Pods with the label app: myapp, the NodePort Service will distribute incoming requests among all three.

Here's what that sentence means in **simple terms**:

☑ Scenario: Pods on Multiple Nodes

- You have **two nodes**:
  - Node 1 → IP 192.168.1.2
  - Node 2 → IP 192.168.1.3
- Your application has **pods running on both nodes**.

☑ What NodePort Does

- When you create a **NodePort Service** (e.g., port 30008):
  - Kubernetes **opens that same port (30008)** on **every node** in the cluster.
- So:
  - Node 1 listens on 192.168.1.2:30008
  - Node 2 listens on 192.168.1.3:30008

☑ Why Is This Important?

- You can use **any node's IP + NodePort** to access your app.
- Example:
  - http://192.168.1.2:30008 → works
  - http://192.168.1.3:30008 → also works
- Kubernetes will **route the request internally** to one of the pods (even if the pod is on a different node).

☑ How Does Routing Work?

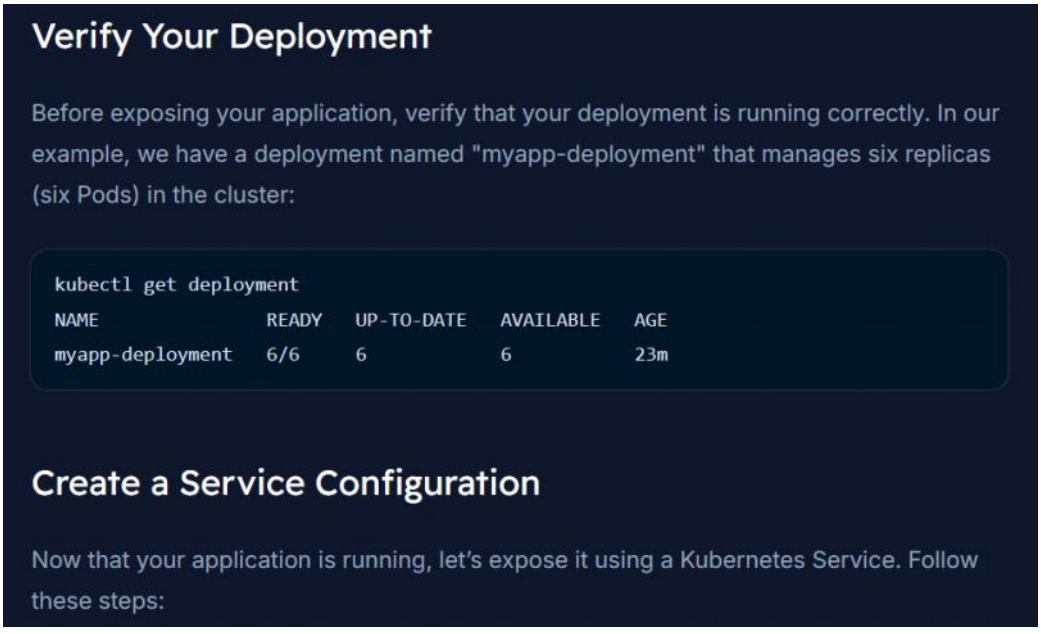
- The NodePort Service knows **all pods** that match its selector.
- If you hit Node 1 but the pod is on Node 2:
  - Node 1 forwards the traffic to Node 2 automatically.
- This is why the service **spans the entire cluster**.

In short:
NodePort makes your app reachable from **any node in the cluster**, using the same port number, regardless of where the pods actually run.

👉 Do you want me to **draw a diagram** showing:

- Two nodes
- NodePort on both
- How traffic flows to pods across nodes?
  Or should I give you a **real-world analogy** (like multiple office buildings with the same reception number)?

In this guide, we'll walk through exposing a Kubernetes application using a Service resource



Demo Services

In this guide, we'll walk through exposing a Kubernetes application using a Service resource. After deploying your application using a Deployment resource, you can make it accessible to users through a web browser by creating a Service.

Verify Your Deployment

Before exposing your application, verify that your deployment is running correctly. In our example, we have a deployment named "myapp-deployment" that manages six replicas (six Pods) in the cluster:

```
kubectl get deployment
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
myapp-deployment  6/6     6            6           23m
```

Create a Service Configuration

Now that your application is running, let's expose it using a Kubernetes Service. Follow these steps:

1. Open your editor and navigate to the directory where you store your configuration files. For better organization, you can create a directory called `service`.
2. Inside the `service` directory, create a file named `service-definition.yaml`.

Directory Structure Flexibility

You are not required to use the suggested directory structure. If preferred, all configuration files may be kept in a single directory.

Define the Service API and Specifications

Begin by defining the API version and kind within your YAML file. Paste in the following configuration:

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  type: NodePort
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30004
```

In this configuration:

- The apiVersion is set to v1 as this is the version used for services.
- The kind is specified as Service.
- The metadata section assigns the service a name: myapp-service.
- Within the spec section:
- The service type is NodePort, which allows external access via a node port (ideal for Minikube).
- The service listens on port 80, mapping it directly to port 80 on the Pods.
- The nodePort is set to 30004 – an allowable value between 30000 and 32767 that exposes the service externally.

Bind the Service to Your Pods

To associate the service with the correct set of Pods, add a selector. Verify that your deployment YAML includes appropriate labels (e.g., app: myapp). Then, update your YAML file with the selector field:

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  type: NodePort
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30004
  selector:
    app: myapp
```

Deploy the Service

Once your service definition is complete, save the file and navigate to its directory to confirm its existence:

```
admin@ubuntu-server service # ls
service-definition.yaml
admin@ubuntu-server service # cat service-definition.yaml
```

Next, create the service in your Kubernetes cluster by executing:

```
kubectl create -f service-definition.yaml
service/myapp-service created
kubectl get svc
```

The output will list your services, including the new NodePort service with its corresponding ClusterIP and port mappings:

```
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes      ClusterIP   10.96.0.1       <none>        443/TCP        24h
myapp-service   NodePort    10.101.76.121   <none>        80:30004/TCP   5s
```

Access Your Application

Since we're using Minikube, retrieve the service URL with:

```
minikube service myapp-service --url
```

This command will output a URL resembling:

```
http://192.168.99.101:30004
```

Copy the URL and paste it into your web browser. You should see the default Nginx web page, confirming that the Nginx application is accessible and running as expected.

Let's make **ClusterIP** super clear:

☑ What is ClusterIP?

- It's the **default Service type** in Kubernetes.
- It creates a **virtual IP address** inside the cluster.
- This IP is **only accessible within the cluster** (not from outside).

☑ Why do we need it?

- Pods have changing IPs.
- If your frontend pod needs to talk to a backend pod, hardcoding pod IPs is unreliable.
- **ClusterIP gives a stable internal address** for communication between pods.

☑ How it works

- Suppose you have a backend service:
  - Pods labeled app: backend
  - Service type: ClusterIP
- Kubernetes assigns a **ClusterIP** like 10.96.0.5.
- Any pod in the cluster can reach the backend using: http://10.96.0.5:
- Or better: using the **service name** (DNS): http://backend-service:

☑ Key Points

- **Internal only** → cannot access from outside the cluster.
- Used for **pod-to-pod communication** (e.g., frontend → backend).
- Kubernetes automatically updates endpoints when pods scale up or down.

☑ YAML Example

```
apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  type: ClusterIP
  ports:
    - port: 8080
      targetPort: 8080
  selector:
```

☑ In short

- ClusterIP = **internal phone number** for your app inside the cluster.
- NodePort = **external phone number** for outside access.
- LoadBalancer = **external phone number with a receptionist** (cloud load balancer).

👉 Do you want me to **draw a simple diagram** showing:

- Pods
- ClusterIP service
- How internal communication works?\ Or should I give you a **real-world analogy** (like office extensions)?

Kubernetes provides **three main types of Services**, each designed for different use cases:

☑ 1. ClusterIP (Default)

- **What it does:** Creates an internal virtual IP inside the cluster.
- **Access:** Only accessible **within the cluster** (pods, nodes).
- **Use case:** Communication between pods (e.g., frontend → backend).
- **Example:** http://backend-service:8080
- **Key point:** Simplifies internal networking.

☑ 2. NodePort

- **What it does:** Exposes the service on a **static port** (30000–32767) on **every node's IP**.
- **Access:** External access via Node-IP:NodePort.
- **Use case:** Access your app from outside the cluster without a load balancer.
- **Example:** http://192.168.1.2:30008
- **Key point:** Works across all nodes; same port everywhere.

☑ 3. LoadBalancer

- **What it does:** Creates an **external load balancer** (in cloud environments).
- **Access:** Provides a **public IP** for your service.
- **Use case:** Production apps needing external access with load balancing.
- **Example:** http://
- **Key point:** Requires cloud provider support (AWS, Azure, GCP).

🔍 Quick Comparison Table

Service Type	Accessible From	Typical Use Case
ClusterIP	Inside cluster	Pod-to-pod communication
NodePort	Outside cluster	Simple external access
LoadBalancer	Outside cluster	Scalable external access

👉 Do you want me to **add the 4th type (ExternalName)** and explain it too?
Or should I **draw a diagram** showing all three types visually?

A Kubernetes service overcomes these challenges by grouping related pods and exposing a single, stable interface. When you create a service for the back-end pods, Kubernetes aggregates them and provides a unified endpoin

NodePort Service Recap

A NodePort service routes external traffic on a specific port to Pods across all nodes, even if the Pods only reside on a subset of those nodes.

Excellent question! Let's break it down:

☑ Why is this a problem?

- A **NodePort Service** opens the same port (e.g., 30008) on **every node** in the cluster.
- If you have multiple nodes, users can access the app using **different node IPs**:
  - http://192.168.1.70:30008
  - http://192.168.1.71:30008
- This creates **multiple URLs** for the same application.
- Users don't know **which node to use**, and if a node goes down, its URL stops working.
- It's confusing and not user-friendly for external access.

☑ Solution

Use a **LoadBalancer Service** (or an Ingress):

- **LoadBalancer Service**:
  - Provides **one external IP** (or DNS name).
  - Distributes traffic across all nodes and pods automatically.
  - Works in cloud environments (AWS, Azure, GCP).
- **Ingress**:
  - Provides a **single entry point** with a domain name.
  - Can handle **routing, SSL, and multiple services** behind one URL.

☑ In short

- **NodePort** = good for testing or small setups, but confusing for users.
- **LoadBalancer or Ingress** = production-ready solution with a single, stable endpoint.

👉 Do you want me to **draw a diagram** comparing:

- NodePort (multiple URLs)
- LoadBalancer (single URL)
- Ingress (single domain with routing)?
  Or should I **write a quick YAML example for LoadBalancer and Ingress**?

Demo Deploying voting app on Kubernetes

tart by creating a new project folder (e.g., "voting-app") and defining the pods for each component in separate YAML files.

1. Voting App Pod

Create a file named voting-app-pod.yaml. This file specifies the API version, kind, metadata (including name and labels), and the container details. The labels group all the components as part of the same application while differentiating each component.

```
apiVersion: v1
kind: Pod
metadata:
  name: voting-app-pod
  labels:
    name: voting-app-pod
    app: demo-voting-app
spec:
  containers:
    - name: voting-app
      image: kodekloud/example-voting-app:_vote-v1
      ports:
        - containerPort: 80
```

2. Result App Pod

Copy the voting app pod template to create the result app pod. Save it as result-app-pod.yaml and update the metadata (name, labels) and container details accordingly.

```
apiVersion: v1
kind: Pod
metadata:
  name: result-app-pod
  labels:
    name: result-app-pod
    app: demo-voting-app
spec:
  containers:
    - name: result-app
      image: kodekloud/examplevotingapp_result:v1
      ports:
        - containerPort: 80
```

3. Redis Pod

For the Redis pod, create a file named redis-pod.yaml. Use the previous template and update the names accordingly. Note that the container port has been changed from 80 to 6379 (the default Redis port).

```
apiVersion: v1
kind: Pod
metadata:
  name: redis-pod
  labels:
    name: redis-pod
    app: demo-voting-app
spec:
  containers:
    - name: redis
      image: redis
      ports:
        - containerPort: 6379
```

4. PostgreSQL (DB) Pod

Using the Redis pod template, create the PostgreSQL pod definition file named postgres-pod.yaml. Update the pod and container names, use the official postgres image, and change the container port to 5432. Additionally, include environment variables for the initial username and password required by the worker and result pods.

```
apiVersion: v1
kind: Pod
metadata:
  name: postgres-pod
  labels:
    name: postgres-pod
    app: demo-voting-app
spec:
  containers:
```

```
    - name: postgres
      image: postgres
      ports:
        - containerPort: 5432
      env:
        - name: POSTGRES_USER
          value: "postgres"
        - name: POSTGRES_PASSWORD
          value: "postgres"
```

5. Worker Pod

Finally, create the worker pod in a file named worker-app-pod.yaml. Use the voting app pod template as a base but update the name and container properties to indicate background processing. Since the worker does not run any service or listen on ports, remove the container port definition.

```
apiVersion: v1
kind: Pod
metadata:
  name: worker-app-pod
  labels:
    name: worker-app-pod
    app: demo-voting-app
spec:
  containers:
    - name: worker-app
      image: kodekloud/examplevotingapp_worker:v1
```

Service Definitions

After defining the pods, create the corresponding services to expose them. All components except the worker require external or internal services.

1. Redis Service (Internal)

Create a file named redis-service.yaml. This service exposes the Redis pod on port 6379 internally, ensuring the selector matches the labels defined in the Redis pod.

```
apiVersion: v1
kind: Service
metadata:
  name: redis
  labels:
    name: redis-service
    app: demo-voting-app
spec:
  ports:
    - port: 6379
      targetPort: 6379
  selector:
    name: redis-pod
    app: demo-voting-app
```

2. PostgreSQL (DB) Service (Internal)

Since the worker expects the Postgres service name to be "DB", create a file named postgres-service.yaml. This service exposes PostgreSQL on port 5432 and selects the appropriate pod.

```
apiVersion: v1
kind: Service
metadata:
  name: db
  labels:
    name: postgres-service
    app: demo-voting-app
spec:
  ports:
    - port: 5432
      targetPort: 5432
  selector:
    name: postgres-pod
    app: demo-voting-app
```

3. Voting App Service (External)

To make the front-end voting app accessible externally, create a service named voting-app-service.yaml. Set the service type to NodePort, expose port 80, and assign a node port, such as 30004. Ensure that the selector matches the labels defined in the voting app pod.

```
apiVersion: v1
kind: Service
metadata:
  name: voting-service
  labels:
    name: voting-service
    app: demo-voting-app
spec:
  type: NodePort
```

```
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30004
  selector:
    name: voting-app-pod
    app: demo-voting-app
```

4. Result App Service (External)

Similarly, create an external service for the result app. Save this file as result-app-service.yaml. This service is also set to type NodePort, exposing port 80 with a node port (e.g., 30005). Update the selector to match the result app pod.

```
apiVersion: v1
kind: Service
metadata:
  name: result-service
  labels:
    name: result-service
    app: demo-voting-app
spec:
  type: NodePort
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30005
  selector:
    name: result-app-pod
    app: demo-voting-app
```

Deploying the Application

With all five pod and service definition files created, navigate to your project directory (e.g., voting-app) and deploy each object using the kubectl create -f command.

```
# Create Pods
kubectl create -f voting-app-pod.yaml
kubectl create -f result-app-pod.yaml
kubectl create -f redis-pod.yaml
kubectl create -f postgres-pod.yaml
kubectl create -f worker-app-pod.yaml
# Create Services
kubectl create -f redis-service.yaml
kubectl create -f postgres-service.yaml
kubectl create -f voting-app-service.yaml
kubectl create -f result-app-service.yaml
```

Verify that all pods and services are active using:

```
kubectl get pods,svc
```

You should see all five pods running, with the voting and result services listed as NodePort services and the Redis and PostgreSQL (DB) services exposed as ClusterIP.

Accessing the Application

To access the voting application, run:

minikube service voting-service --url

This opens the front-end interface where you can cast your vote.

Once you cast a vote, check the results by accessing the result service:

```
minikube service result-service --url
```

This will display the updated voting results.

Explanation

Here's a **lucid explanation** of the Kubernetes deployment process for the voting app:

☑ What Are We Doing?

We are deploying a **microservices-based voting application** on Kubernetes.\ It has **five components**:

1. **Voting App (Python)** → Frontend for voting.
2. **Result App (Node.js)** → Shows results.
3. **Redis** → Temporary storage for votes.
4. **PostgreSQL** → Permanent database.
5. **Worker (.NET)** → Processes votes and updates DB.

☑ Step 1: Create Pods

Each component runs in its own **Pod**.\ We write **YAML files** for each pod:

- **Voting App Pod** → Runs on port 80.
- **Result App Pod** → Runs on port 80.
- **Redis Pod** → Runs on port 6379.
- **Postgres Pod** → Runs on port 5432 (with username/password env vars).
- **Worker Pod** → No exposed port (background job).

☑ Step 2: Create Services

Pods need **Services** for communication:

- **Redis Service (ClusterIP)** → Internal access only.
- **Postgres Service (ClusterIP)** → Internal access only.
- **Voting App Service (NodePort)** → External access (port 30004).
- **Result App Service (NodePort)** → External access (port 30005).

Why?

- ClusterIP = internal communication (pods talk to DB/Redis).
- NodePort = external access (users access frontend & results).

☑ Step 3: Deploy

Run:

```
kubectl create -f voting-app-pod.yaml
kubectl create -f result-app-pod.yaml
kubectl create -f redis-pod.yaml
kubectl create -f postgres-pod.yaml
kubectl create -f worker-app-pod.yaml
kubectl create -f redis-service.yaml
kubectl create -f postgres-service.yaml
kubectl create -f voting-app-service.yaml
kubectl create -f result-app-service.yaml
```

Check:

```
kubectl get pods,svc
```

You'll see:

- **5 pods running**
- **4 services** (2 NodePort, 2 ClusterIP)

☑ Step 4: Access the App

If using **Minikube**:

minikube service voting-service --url
minikube service result-service --url

- First URL → Voting page (cast vote).
- Second URL → Results page (see updated counts).

☑ How It Works Internally

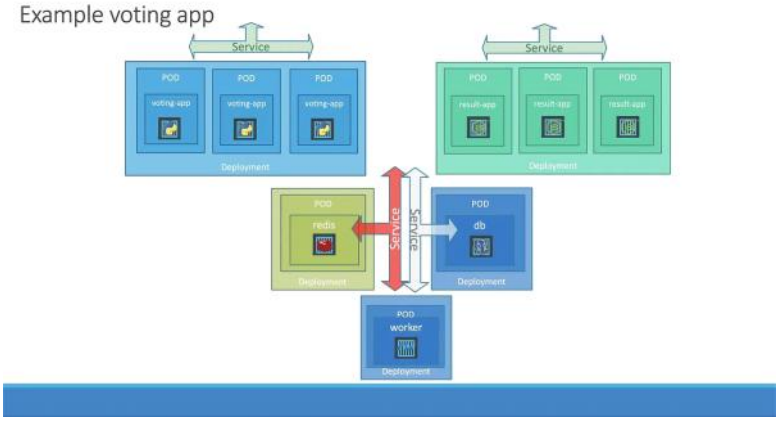
- Voting App → sends vote to Redis.
- Worker → reads from Redis, updates PostgreSQL.
- Result App → reads from PostgreSQL, shows results.
- Kubernetes Services ensure **pods find each other by name** (DNS), not IP.

👉 Do you want me to **draw a diagram** showing:

- All pods
- Services (ClusterIP & NodePort)
- How traffic flows from user → voting app → Redis → worker → PostgreSQL → result app?\ Or should I **give you a single combined YAML file** for the entire stack?

Demo Deploying voting app on Kubernetes with Deployments

In this lesson, we enhance the initial demo—which deployed Pods and Services directly—by leveraging Kubernetes Deployments. This improved approach addresses the challenges of scaling and updating applications without downtime. By using Deployments, you can automate the management of ReplicaSets, making it simple to scale, roll out updates, and perform rollbacks while retaining a history of revisions.



Note

Deploying individual Pods limits your ability to easily increase the number of service instances or update the container image without downtime. Using Deployments streamlines these processes.

In the upgraded setup, each application component—including the front-end apps (voting and results), databases, and worker applications—is encapsulated within its own Deployment. The project directory now hosts both the original Pod and Service definition files, along with new files for the Deployments.

Voting App Deployment

We begin by defining a basic Pod for the voting app:

```
apiVersion: v1
kind: Pod
metadata:
  name: voting-app-pod
  labels:
    name: voting-app-pod
    app: demo-voting-app
spec:
  containers:
    - name: voting-app
      image: kodekloud/examplevotingapp_vote:v1
      ports:
        - containerPort: 80
```

Then, create a Deployment file (votingapp-deployment.yaml) based on this pod template. Adjust the API version to apps/v1, change the kind to Deployment, and add the selector and replicas fields:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: voting-app-deploy
  labels:
    name: voting-app-deploy
    app: demo-voting-app
spec:
  replicas: 1
  selector:
    matchLabels:
      name: voting-app-pod
      app: demo-voting-app
  template:
    metadata:
      name: voting-app-pod
      labels:
        name: voting-app-pod
        app: demo-voting-app
    spec:
      containers:
        - name: voting-app
          image: kodekloud/examplevotingapp
          ports:
            - containerPort: 80
```

This configuration allows you to start with a single replica on your cluster for resource efficiency, with the option to scale up as needed.

Redis Deployment

Begin with the original Redis Pod definition:

```
# redis-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: redis-pod
  labels:
    app: demo-voting-app
spec:
  containers:
    - name: redis
      image: redis
      ports:
        - containerPort: 6379
```

Then, create the Redis Deployment (redis-deploy.yaml) using the same template as the voting app deployment. Update the component names, labels, and container details accordingly:

```
# redis-deploy.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  name: redis-deploy
  labels:
    name: redis-deploy
    app: demo-voting-app
spec:
  replicas: 1
  selector:
    matchLabels:
      name: redis-pod
      app: demo-voting-app
  template:
    metadata:
      name: redis-pod
      labels:
        name: redis-pod
        app: demo-voting-app
    spec:
      containers:
        - name: redis
          image: redis
          ports:
            - containerPort: 6379
```

PostgreSQL Deployment

Transform the PostgreSQL Pod into a Deployment. First, consider the original PostgreSQL Pod definition with environment variables:

```
apiVersion: v1
kind: Pod
metadata:
  name: postgres-pod
  labels:
    name: postgres-pod
    app: demo-voting-app
spec:
  containers:
    - name: postgres
      image: postgres
      ports:
        - containerPort: 5432
      env:
        - name: POSTGRES_USER
          value: "postgres"
        - name: POSTGRES_PASSWORD
          value: "postgres"
```

Now, create the PostgreSQL Deployment (postgres-deploy.yaml). Ensure that the selector matches the Pod template labels:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres-deploy
  labels:
    name: postgres-deploy
    app: demo-voting-app
spec:
  replicas: 1
  selector:
    matchLabels:
      name: postgres-pod
      app: demo-voting-app
  template:
    metadata:
      name: postgres-pod
      labels:
        name: postgres-pod
        app: demo-voting-app
    spec:
      containers:
        - name: postgres
          image: postgres
          ports:
            - containerPort: 5432
          env:
            - name: POSTGRES_USER
              value: "postgres"
            - name: POSTGRES_PASSWORD
              value: "postgres"
```

Worker App Deployment

For the worker application, start with its Pod definition:

```
apiVersion: v1
kind: Pod
metadata:
  name: worker-app-pod
  labels:
    name: worker-app-pod
    app: demo-voting-app
spec:
  containers:
    - name: worker-app
      image: kodekloud/examplevotingapp
```

Then, create a corresponding Deployment (worker-app-deploy.yaml). Update the names, labels, and selectors as required:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: worker-app-deploy
  labels:
    name: worker-app-deploy
    app: demo-voting-app
spec:
  replicas: 1
  selector:
    matchLabels:
      name: worker-app-pod
      app: demo-voting-app
  template:
    metadata:
      name: worker-app-pod
      labels:
        name: worker-app-pod
        app: demo-voting-app
    spec:
      containers:
        - name: worker-app
          image: kodekloud/examplevotingapp
```

Result App Deployment

Similarly, convert the result application from a Pod to a Deployment. Create the file result-app-deploy.yaml, update the names and labels from the original Pod definition, and ensure that the template matches the selector criteria.

Your project directory should now include files similar to the following:

```
voting-app-pod.yaml
result-app-pod.yaml
redis-pod.yaml
postgres-pod.yaml
redis-service.yaml
postgres-service.yaml
voting-app-service.yaml
result-app-service.yaml
worker-app-pod.yaml
voting-app-deploy.yaml
redis-deploy.yaml
postgres-deploy.yaml
worker-app-deploy.yaml
result-app-deploy.yaml
```

Deploying on Kubernetes

Before creating the new Deployments and Services, ensure that any previously created resources are removed. Verify by running:

```
kubectl get pods
```

Once the cluster is clean, proceed with the deployments.

Creating the Voting App Deployment

Deploy the voting app using the command below:

```
kubectl create -f voting-app-deploy.yaml
```

Verify the deployment:

```
kubectl get deployment
```

Expected output:

```
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
voting-app-deploy   1/1     1            1           19s
```

Deploying Redis, PostgreSQL, and Their Services

First, deploy Redis and its service:

```
kubectl create -f redis-deploy.yaml
kubectl create -f redis-service.yaml
```

Then, deploy PostgreSQL and its service:

```
kubectl create -f postgres-deploy.yaml
kubectl create -f postgres-service.yaml
```

Confirm that all Pods are running:

```
kubectl get pods
```
Sample output:
```
NAME                                READY   STATUS    RESTARTS   AGE
postgres-deploy-847c9c8d8f-dzk8m    1/1     Running   0          19s
redis-deploy-5b479fbfd5-ndxbn       1/1     Running   0          27s
voting-app-deploy-7775f98f7d-2xdlz  1/1     Running   0          56s
```
Deploying Worker and Result Applications
Deploy the worker application (which does not have an associated Service):
```
kubectl create -f worker-app-deploy.yaml
```
Next, deploy the result application and its service:
```
kubectl create -f result-app-deploy.yaml
kubectl create -f result-app-service.yaml
```
Finally, check that all Deployments and Services are active:
```
kubectl get deployments,svc
```
Example output:
```
NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/postgres-deploy     1/1     1            1           91s
deployment.apps/redis-deploy        1/1     1            1           99s
deployment.apps/result-app-deploy   1/1     1            1           18s
deployment.apps/voting-app-deploy   1/1     1            1           2m8s
deployment.apps/worker-app-deploy   1/1     1            1           45s
NAME                     TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
service/db               ClusterIP   10.107.65.177   <none>        5432/TCP       88s
service/kubernetes       ClusterIP   10.96.0.1       <none>        443/TCP        2d1h
service/redis            ClusterIP   10.104.71.94    <none>        6379/TCP       95s
service/result-service   NodePort    10.105.105.132  <none>        80:3005/TCP    15s
service/voting-service   NodePort    10.100.70.146   <none>        80:3004/TCP    119s
```

Accessing the Front-End Applications
Use Minikube to retrieve the URLs for the voting and result services:
```
minikube service voting-service --url
minikube service result-service --url
# Expected output: http://192.168.99.101:30005
```
Open these URLs in your web browser to interact with the voting app. Load balancing within the Deployment ensures that different pods serve your requests.

Scaling the Voting App Deployment
Scaling the voting application is straightforward with Deployments. For example, to increase the number of voting app replicas from one to three, run:
```
kubectl scale deployment voting-app-deploy --replicas=3
```
After scaling, verify the status:
```
kubectl get deployments
```
Refresh the voting service URL in your browser several times to observe that requests are being handled by multiple pods, confirming that the scaling is effective.