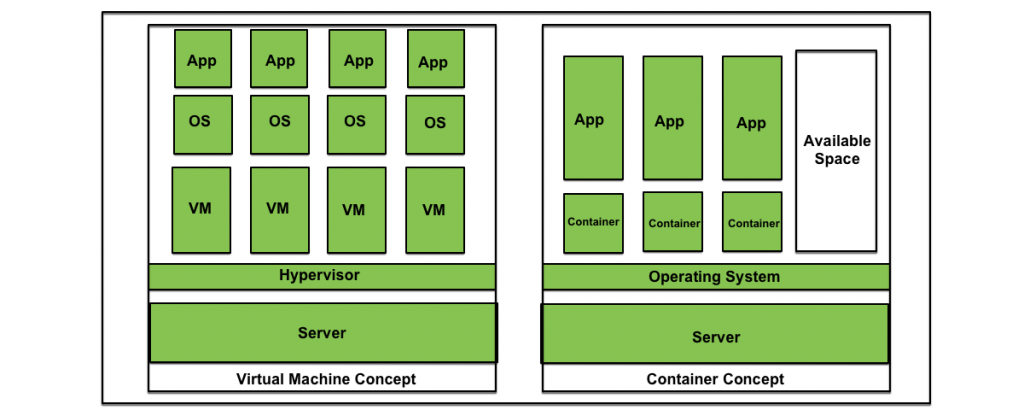
**Docker Fundamentals**

**What is a container?**

container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings.



**1. Lightweight**

**Containers: They don’t need a full operating system, so they’re smaller and start faster.**

* **VMs: Each VM has its own OS, making them bigger and slower to start.**

**2. Efficient**

* **Containers: Share resources like the OS kernel, so they use less CPU and memory.**
* **VMs: Need fixed resources, even if they’re not fully used, which can waste resources.**

**3. Fast Scaling**

* **Containers: Start in seconds, so you can quickly add or remove them to handle more work.**
* **VMs: Take longer to set up and scale because of their size.**

**4. Easy to Move**

* **Containers: You can package your app with everything it needs and run it anywhere.**
* **VMs: Harder to move because they depend on specific setups.**

**5. Simple Updates**

* **Containers: Updating is easy—just update the container and restart it.**
* **VMs: You have to update the whole OS inside the VM, which takes more time.**

**When to Use:**

* **Containers: Best for quick, efficient, and portable applications (like modern apps or microservices).**
* **VMs: Better for tasks that need full isolation or different operating systems.**

**Why container is lightweight**

* Containers are small and efficient because they share the operating system of the host machine instead of including a full OS for each container.
* They only have the application and its needed files, which keeps them lightweight and fast.
* This way, you can run more containers on the same machine without using too many resources.

**Files and Folders in containers base images**

/bin: contains binary executable files, such as the ls, cp, and ps commands.

/sbin: contains system binary executable files, such as the init and shutdown commands.

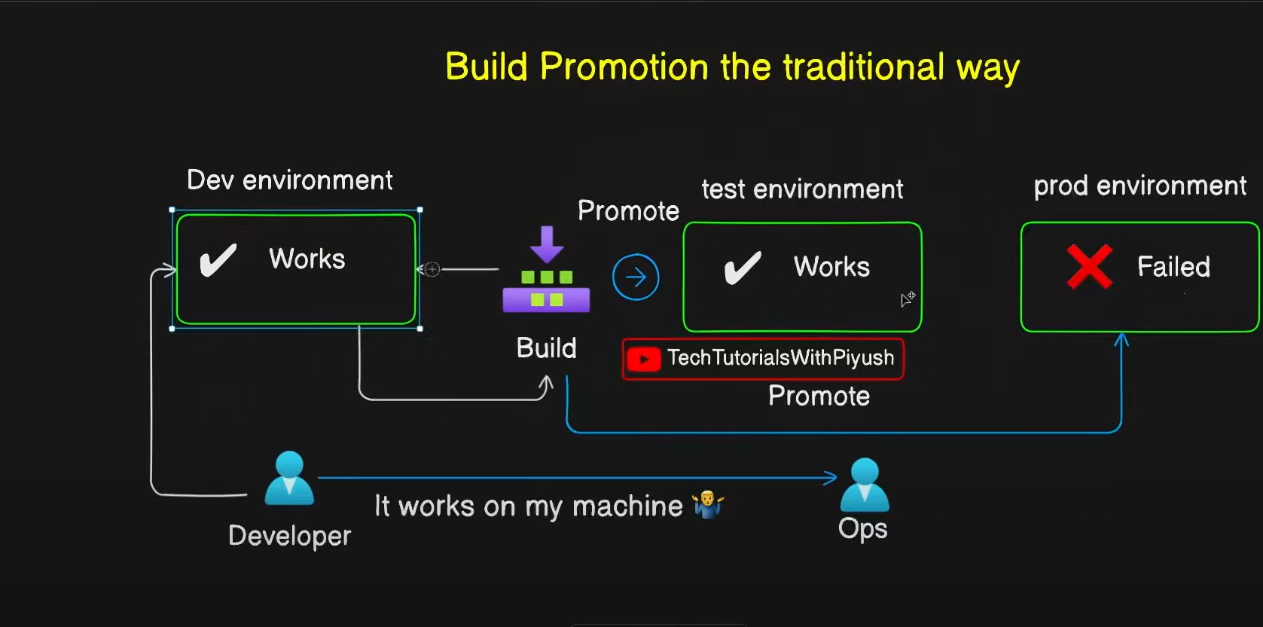
/etc: contains configuration files for various system services.

/lib: contains library files that are used by the binary executables.

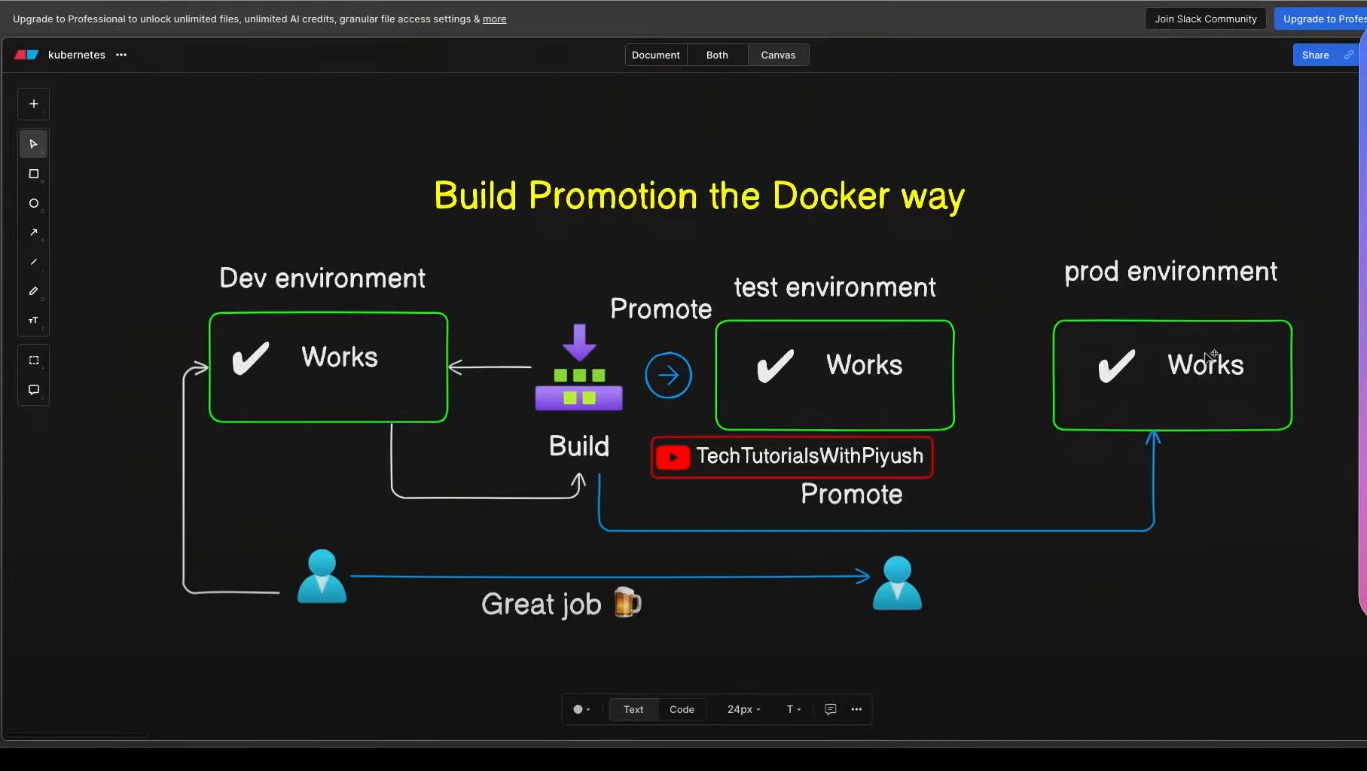
/usr: contains user-related files and utilities, such as applications, libraries, and documentation.

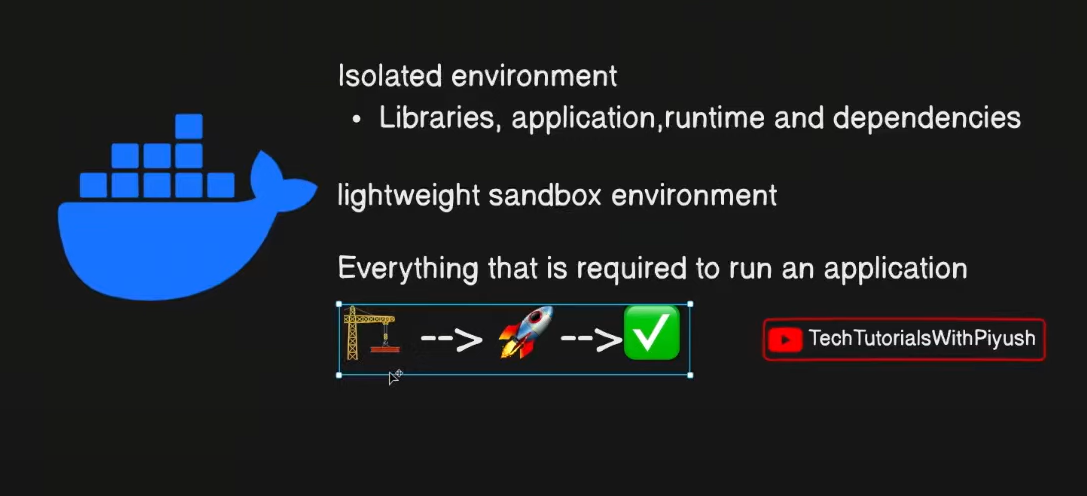
/var: contains variable data, such as log files, spool files, and temporary files.

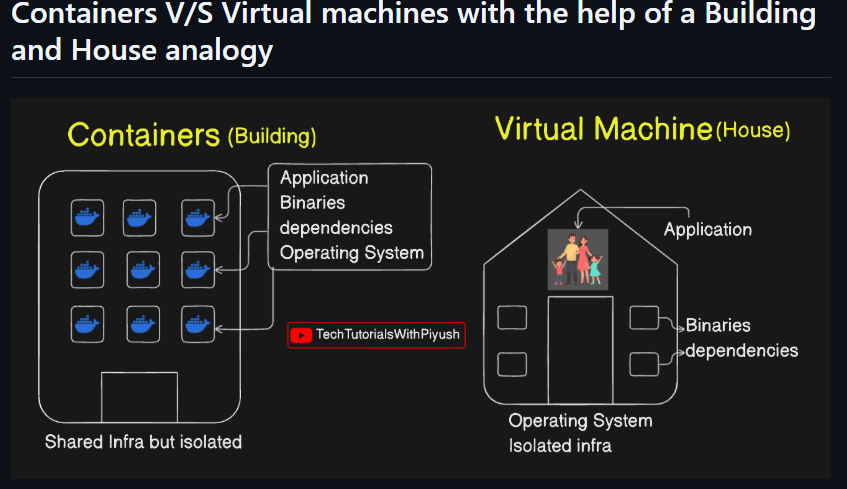
/root: is the home directory of the root user.

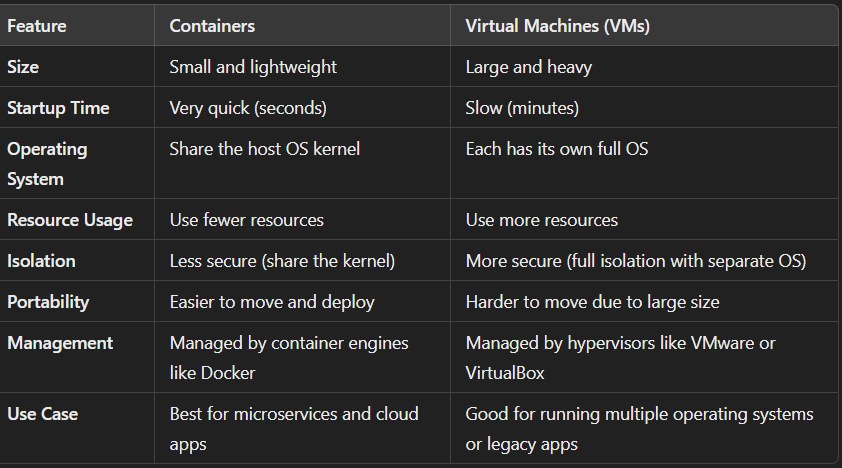
****

* Before containers, applications often failed in production due to **misconfigurations and missing dependencies**. Containers solve this problem by packaging everything the application needs, **including its code**, **libraries**, **dependencies**, and parts of the **operating system.**

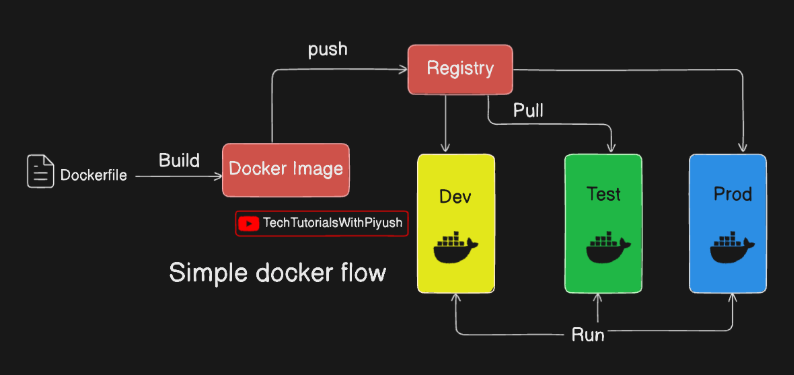
****

****



****

**The flow docker files**

****

* **Build**
* Start with a Dockerfile, which contains instructions for creating a Docker image.
* The docker build command uses the Dockerfile to create an image that includes the operating system, libraries, and application code**.**
* This file is in the form of binary format

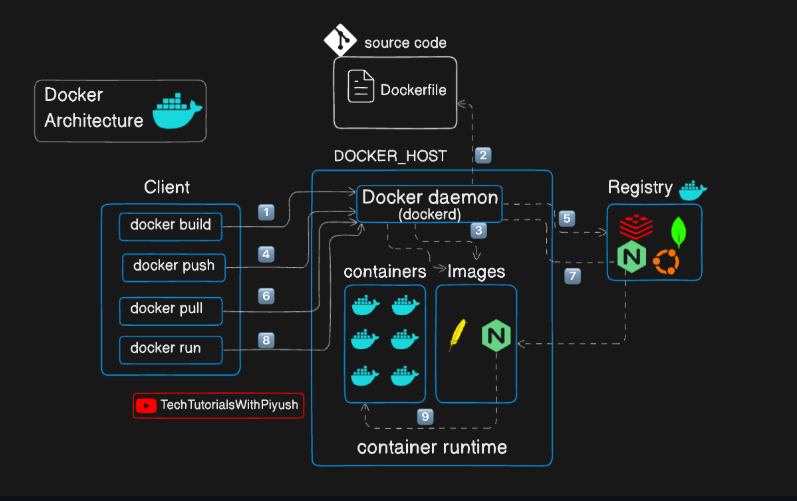
**2. Ship**

* The Docker image is uploaded to a Docker registry, a storage system for sharing images.
* This is similar to how a version control system like GitHub stores source code
* Public registries like Docker Hub or private ones like Artifactory or Nexus are used.
* The docker push command sends the image to the registry

**3 .Run: Run:**

To deploy an application, the Docker image is pulled from the registry to the target environment, such as development, testing, or production2. This is done using the docker pull command2.

Finally, the docker run command creates a running container instance based on the pulled image2. This container executes the application with all its bundled dependencies



**Docker file**

**How To Dockerize a Project**

**Step 1: Install Docker**

* Download and install **Docker Desktop** from the Docker website.
* Alternatively, use the online sandbox environment at **play-with-docker.com**

**Step 2: Prepare the Application**

* **mkdir <file1 name >**
* **cd <file1 name>**
* Obtain the application code by cloning a repository or using a local project:
* git clone <https://github.com/docker/getting-started-app.git>

tting-started-app/

* cd getting-started-app/

**Step 3: Create a Dockerfile**

* touch Dockerfile
* vi to the file

**Step 4: Write Instructions in the Dockerfile**

FROM node:18-alpine

WORKDIR /app

COPY . .

RUN yarn install --production

CMD ["node", "src/index.js"]

EXPOSE 3000

**FROM node:18-alpine**

* Purpose: Sets the **base image** for the Docker build**.**
* we can select an OS but this very weightless
* It contains the necessary runtime and dependencies for Node.js applications.

**WORKDIR /app**

* **Purpose:** Sets the working directory inside the Docker container**.**

**COPY . .**

* **Purpose**: Copies the files from the current directory on the host to the working directory

**RUN yarn install –production**

* Yarn is a package manager
* Purpose: Installs the dependencies for the project**.**

**CMD ["node", "src/index.js"]**

* Purpose: Specifies the command to run when a container is started from the image.
* Details: The application starts by running the index.js file located in the src directory using Node.js**.**

**EXPOSE 3000**

**Purpose:** Documents that the application listens on port 3000.

**Docker build**

* **Build the Docker Image**

docker build -t day2-todo.

* . current working dir
* **Tag the image** for Docker Hub:

docker tag local-image:tagname new-repo:tagname

docker tag day2-todo:latest **gopichand2/test-repo:latest**

* **Docker login**
* Docker push **gopichand2/test-repo:latest**
* **Docker run**
* **docker run -dp 3000:3000 username/new-reponame:tagname**

**Docker Multi-Stage Builds**

* **Image**: A lightweight, standalone, and executable package that contains all the code, libraries, and settings needed to run an application.
* **Container**: A running instance of a Docker image.
* **Multi-Stage Build**: A Dockerfile technique where multiple temporary build stages are used to create a smaller, more efficient final image.
* **Stage**: A step in the Dockerfile with its own base image and set of command
* **Step-by-Step Guide**

**Clone the below sample repository**

* git clone <https://github.com/piyushsachdeva/todoapp-docker.git>

**cd into the directory**

**create docker file**

* touch Docker file
* vi Docker file

**FROM node:18-alpine AS installer**

* As installer is name of the stage

FROM node:18-alpine AS installer

WORKDIR /app

COPY package\*.json ./

RUN npm install

COPY . .

RUN npm run build

FROM nginx:latest AS deployer

COPY --from=installer /app/build /usr/share/nginx/html

**Save the file and run docker build command**

* Docker build -t multi-stage .
* <Multi -stage > name of the build

**Docker run command**

* docker run -it -dp 3000:3000 multi-stage

**Docker command**

 **docker build**: Builds a Docker image from a Dockerfile. Use -t to tag it (e.g., docker build -t my-image .).

 **docker images**: Lists all local Docker images with name, tag, and size.

 **docker run**: Runs a container from an image. Use -d for detached mode and -p to map ports (e.g., docker run -d -p 8080:80 my-image).

 **docker ps**: Lists currently running containers.

 **docker stop**: Stops a running container by ID or name.

**docker exec**: Runs commands inside a running container (e.g., docker exec -it <container ID> bash).

* docker exct -it container id sh
* pwd
* ls-lrt
* cd var/log
* cd nginx
* cd ../
* cd /user/share/nginx/html

 **docker rm**: Removes a stopped container by ID or name.

 **docker image rm**: Deletes an image by ID or name to free space.

 **docker exec**: Runs commands inside a running container (e.g., docker exec -it <container ID> bash).

 **docker logs**: Shows logs from a container (e.g., docker logs <container ID>).

 **docker inspect**: Provides detailed information about a container or image for troubleshooting.

A **multi-stage Docker build** solves the problem of large image sizes and improves efficiency and security. Here's how:

* **Smaller Image Size**: It uses a "builder" stage to compile code and install dependencies, then copies only necessary files to the final image, making it smaller.
* **Faster Performance**: Small images download and start quickly, using fewer resources.
* **Better Security**: Only required files are included, leaving out sensitive files and development tools, reducing vulnerabilities.
* **Isolation**: Each stage is separate, so unnecessary files don’t get into the final image.
* **Optimized for Production**: The final image contains only what's needed to run the app, not extra build tools.

**What is Kubernetes?**

Kubernetes (K8s) is a tool that helps you manage apps running in **containers** (small, isolated environments for your apps). It organizes, scales, and keeps them running smoothly. Think of Kubernetes as a **traffic controller** for your apps, making sure they don’t crash, work fast, and are always available to users.

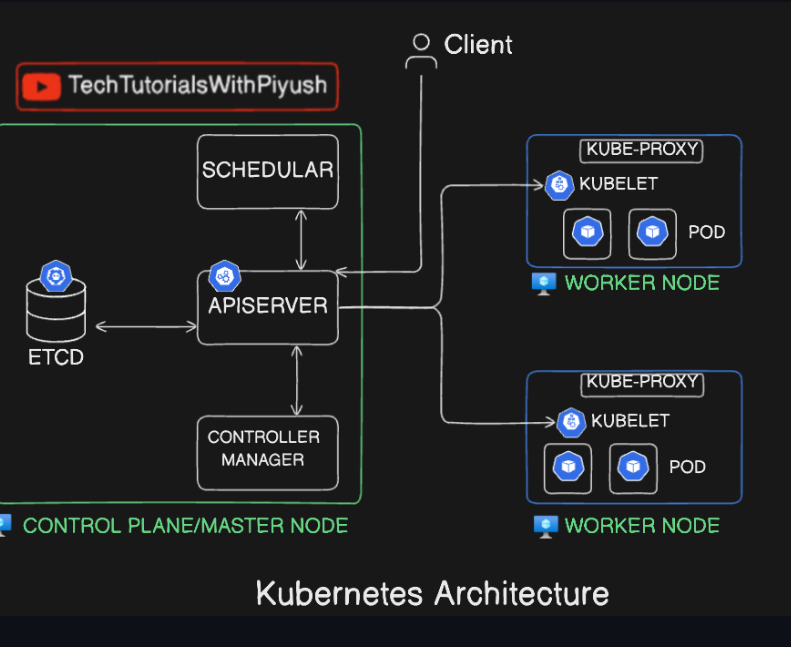
**Key Terms in Simple Language**

* **Container**: A small package that includes an app and everything it needs to run. Like a lunchbox with food and utensils all packed together.
* **Pod**: The smallest unit in Kubernetes, usually containing one or more containers that work together.
* **Node**: A computer (real or virtual) where the containers run.
* **Cluster**: A group of nodes working together, managed by Kubernetes.
* **Deployment**: A set of instructions that tells Kubernetes how to manage and run your containers.
* **Service**: Helps users find and use your app by managing the connection between the app and the internet.

**Kubernetes Architecture Overview**

* In Kubernetes, a node is a machine (physical or virtual) that performs work by running containerized applications.
* Each cluster consists of two types of nodes

1. Control planes /master node
2. Worker node



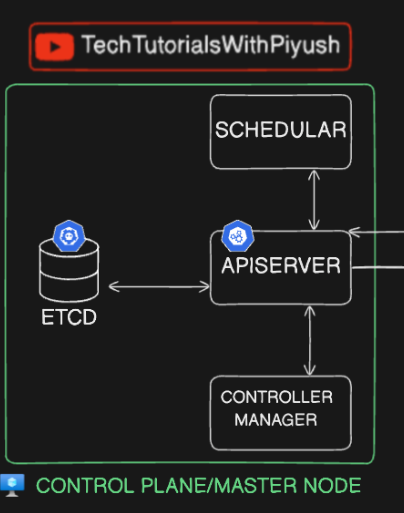
**1. Control Plane (Master) Node**

The control plane node manages and controls the entire Kubernetes cluster. It makes scheduling decisions, handles cluster management, and maintains the desired state of the applications.

**Key Components of the Control Plane Node**:

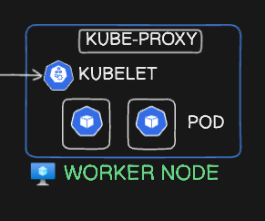
* **API Server**: The entry point for all cluster requests.
* **Scheduler**: Assigns pods to worker nodes.
* **Controller Manager**: Ensures resources match the desired state.
* **etcd**: A key-value store that keeps the cluster’s state information.

It is stored in JSON format



**2. Worker Nodes**

* Worker nodes are responsible for running the actual application workloads. They are managed by the control plane and host the containers inside **pods**.
* **Key Components of Worker Nodes**:
* **Kubelet**: Communicates with the control plane to manage pods and containers on the node.
* **Kube-proxy**: Manages networking to allow communication between pods and external systems.
* **Container Runtime**: The software (like Docker or containers) that runs the containers

****

**Pod**

* a pod is the smallest and simplest unit that can be deployed, managed, and scaled.
* It is an abstraction layer for one or more containers that share the same network, storage, and resources.
* Pods allow containers within them to work together as a single unit
* Pod help to run the container

**flow of Kubernetes architecture**:

Here’s a simple flow of Kubernetes architecture:

1. **User Sends a Request**:
   * A user or DevOps engineer uses a tool like kubectl to create or manage resources (e.g., create a pod).
2. **API Server Receives the Request**:
   * The **API Server** is the central point for all requests. It authenticates and validates the request.
3. **etcd Stores the Cluster State**:
   * The API Server updates **etcd**, which stores all the cluster data and state information.
4. **Scheduler Assigns the Pod to a Node**:
   * If a new pod needs to be created, the **Scheduler** finds the best worker node based on available resources and scheduling rules.
5. **API Server Instructs the Kubelet**:
   * The **API Server** communicates with the **Kubelet** on the selected worker node to create the pod.
6. **Kubelet Manages Pods on the Worker Node**:
   * The **Kubelet** runs the pod and ensures the containers are healthy and functioning.
7. **Kube-proxy Handles Networking**:
   * The **Kube-proxy** enables networking between pods and external systems.
8. **Controller Manager Ensures Desired State**:
   * The **Controller Manager** monitors resources and makes sure the system matches the desired state (e.g., correct number of pods running).

**Kubernetes Local Cluster Setup with Kind**

**Introduction to Kubernetes and Kind**

* **Kubernetes**: An open-source platform for managing containerized applications.
* **Kind (Kubernetes IN Docker):** A tool to run local Kubernetes clusters using Docker container nodes.

**Windows**:

* choco install kind

**docker images --**no-trunc (Simple Explanation)

The docker images command lists all Docker images on your system.

it shortens (truncates) the Image ID to the first 12 characters.

The --no-trunc flag shows the full Image ID (64 characters) instead of a shortened version.

**docker images -a**

Show all images (default hides intermediate images)

**List images by name and tag**

[REPOSITORY[:TAG]]

**Saving Images and Containers as Tar Files for Sharing**

* The docker export - Export a container’s filesystem as a tar archive

A **TAR archive** (.tar file) is a **compressed or uncompressed file format** used to store multiple files in a single file. It is commonly used for backups, packaging files, and transferring large sets of data.

* The docker import - Import the contents from a tarball to create a filesystem image
* The docker save - Save one or more images to a tar archive (streamed to STDOUT by default)
* The docker load - Load an image from a tar archive or STDIN

**Create Nginx Container**

docker run -d -p 80:80 nginx

**-d** → Runs the container in the background.

**-p** → Maps a host port to a container port.