🧠 I. What is Kubernetes?

Kubernetes (a.k.a. K8s) is an **open-source platform** for **automating deployment, scaling, and management** of **containerized applications**.

Think of it as the **"operating system for the cloud."**

🚢 Why Kubernetes Exists

Before Kubernetes, developers faced pain points:

Problem	Kubernetes Solution
Deploying many containers manually is hard	Automates deployment
Apps crash or servers fail	Auto-healing restarts
Need to scale up/down fast	Auto-scaling
Load balancing traffic	Built-in load balancing
Configuration drift between dev/stage/prod	Declarative configs via YAML

In short, Kubernetes ensures your app runs **reliably, everywhere, all the time.**

⚙️ Core Concepts

Let's break down the key building blocks.

🧩 1. Cluster

A **Kubernetes Cluster** is a group of computers (nodes) working together to run your applications.

It has two parts:

- **Control Plane** — the brain 🧠 (manages everything)
- **Worker Nodes** — the muscles 💪 (run your apps)

🧠 2. Control Plane Components

Component	Function
API Server	The front door — you talk to it via `kubectl` or the dashboard
etcd	A distributed key-value store that remembers the cluster state
Controller Manager	Ensures the actual state matches the desired state
Scheduler	Decides which node runs which workload

These ensure everything runs as you *declared* in YAML files.

🖥️ 3. Worker Node Components

Each node runs:

Component	Function
kubelet	Talks to the API Server, manages Pods on this node
kube-proxy	Handles network routing and load balancing
Container Runtime	Runs containers (e.g., Docker, containerd, CRI-O (Container Runtime Interface – Open))

📦 4. Pods: The Smallest Deployable Unit

A **Pod** is the **basic unit** in Kubernetes — it's a wrapper around one or more containers that must run together.

- 1 Pod = 1 Application instance
- Each Pod has its own IP address
- If a Pod dies, Kubernetes creates a new one automatically

💡 Analogy:
If containers are processes, Pods are like "logical hosts" for those processes.

🧰 5. Controllers: Keeping Desired State

Controllers manage Pods automatically.

Controller	Description
ReplicaSet	Ensures a fixed number of Pods are always running
Deployment	Manages updates and rollbacks for ReplicaSets
DaemonSet	Runs one Pod on every node (e.g., logging agents)
StatefulSet	Manages stateful apps (like databases)
Job / CronJob	Run tasks once or on a schedule

🌍 6. Services: Stable Networking

Pods come and go, but a **Service** gives them a **stable IP and DNS name**.

Type	Description
ClusterIP	Internal access only
NodePort	Exposes app on a port of each node
LoadBalancer	Integrates with cloud load balancer (AWS, GCP, etc.)
ExternalName	DNS-based redirection to external service

💡 Think of a Service as the "front door" to a set of Pods.

⚡ 7. Ingress: Web Traffic Gateway

Ingress manages HTTP and HTTPS traffic into your cluster.
It routes requests like a reverse proxy (similar to Nginx or Traefik).

Example:

```
/api -> user-service
/web -> frontend-service
```

🔐 8. ConfigMaps & Secrets

Resource	Purpose
ConfigMap	Stores configuration (non-sensitive)
Secret	Stores sensitive data (passwords, API keys, etc.)

These are injected into Pods as environment variables or files.

📈 9. Scaling & Healing

Kubernetes constantly compares the **desired state** (what's declared in YAML) vs. **actual state** (what's running).

If something fails:

- It restarts Pods automatically.
- It can scale Pods up/down based on CPU, memory, or custom metrics.

Example:

```
kubectl scale deployment myapp --replicas=5
```

🧾 10. YAML Files — The Declarative Magic

You tell Kubernetes *what* you want, not *how* to do it.

Example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp
        image: nginx:latest
        ports:
        - containerPort: 80
```

Then you apply it:

`kubectl apply -f myapp.yaml`
Kubernetes will make sure three `nginx` Pods are always running.

☁️ 11. Cloud-Native Integration

Kubernetes is designed to run *anywhere*:

- On-premises
- AWS, GCP, Azure
- Your laptop (via Minikube, Docker Desktop, Kind)

It's the same abstraction layer across all environments — that's its power.

🧩 12. Add-ons & Ecosystem

Kubernetes is just the core — the ecosystem is massive.

Tool	Purpose
Helm	Package manager for Kubernetes apps (like apt/yum for clusters)
Prometheus + Grafana	Monitoring and dashboards
Istio / Linkerd	Service mesh (traffic control, security, observability)
ArgoCD / Flux	GitOps (auto-deploy from Git)
Kustomize	YAML templating for environments

🚀 13. How Deployments Work in Practice

Example flow:

1. Developer writes code → builds a Docker image → pushes to registry
2. Creates a YAML Deployment file
3. Runs `kubectl apply -f deployment.yaml`
4. Kubernetes pulls the image, creates Pods, exposes them via Service
5. LoadBalancer/Ingress routes user traffic
6. Autoscaling + health checks keep it reliable

🧭 14. Mental Model Summary

Concept	Real-World Analogy
Cluster	Data center
Node	Machine/server
Pod	App instance
Container	Process
Service	Load balancer
Deployment	App version manager
ConfigMap/ Secret	Environment variables
Ingress	Router/reverse proxy

🧠 TL;DR Summary

Kubernetes = “automated operations for containers.”

It:

- Deploys your app
- Keeps it alive
- Scales it
- Routes traffic
- Handles secrets
- Works anywhere

🧩 II. What does “Integrating Kubernetes and Docker” mean?

At a high level:

Docker builds and packages your app → Kubernetes deploys and manages those packages at scale.

So “integration” means:

- Using Docker to **containerize** your app
- Using Kubernetes to **run, scale, and manage** those containers automatically

They’re not competing tools — they’re parts of a full pipeline:

Stage	Tool	What happens
Develop & build	🐳 Docker	You build and test your app as an image
Store & share	Docker Registry (Docker Hub or private)	You push/pull the image
Run & scale	☸️ Kubernetes	You define how many containers run, how they talk to each other, and expose them

So “integrating” them means:
Docker is your **build system**, Kubernetes is your **orchestrator**.

⚙️ 2. Architecture overview

When you integrate Docker and Kubernetes, your workflow looks like this:

```
[Source code + Dockerfile] → docker build → docker push → kubectl apply (uses image)
```

In a cluster, the structure looks like:

```
Kubernetes Master (control plane)
     ├── Schedules Pods
     ├── Decides placement
     └── Talks to container runtime (Docker / containerd / CRI-O)
Kubernetes Nodes (workers)
     └── Run your containers using the runtime (Docker)
```

Docker (or containerd) is the runtime that actually starts containers on each node.
Kubernetes tells Docker *when* and *how* to run them.

🔧 3. How Kubernetes uses Docker under the hood

Originally, Kubernetes used Docker directly to start containers (via **dockershim**).
Now it uses **containerd** — which is part of Docker — through the **Container Runtime Interface (CRI)**.

So even though Kubernetes no longer directly depends on Docker, your **Docker-built images** still work perfectly because:

- Docker builds **OCI-compliant images**
- Kubernetes can run **any OCI image**, regardless of the runtime

✅ **So Docker and Kubernetes are compatible by design.**

🪜 4. Practical Integration Steps (on your system)

Let's do it step by step with your `event_registration.html` example.

Step 1 — Containerize your app using Docker

We'll use this Dockerfile (you already built this earlier):

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

Then build the image:

```
docker build -t event-app:1.0 .
```

✅ You now have a Docker image named `event-app:1.0`.

Step 2 — Verify Docker works

Run it manually:

```
docker run -d -p 8080:80 event-app:1.0
```

Go to: http://localhost:8080
You should see your HTML page.

Stop the container:

```
docker stop $(docker ps -q --filter ancestor=event-app:1.0)
```

Step 3 — Push the image (optional)

If your Kubernetes cluster can't access your local image (like on cloud K8s), push it to Docker Hub:

```
docker tag event-app:1.0 <your-dockerhub-username>/event-app:1.0
docker login
docker push <your-dockerhub-username>/event-app:1.0
```

If you're using **Minikube** or **Docker Desktop**, you can skip pushing and just load it locally.

For Minikube:

```
minikube image load event-app:1.0
```

Step 4 — Create your Kubernetes manifest

Create a `k8s.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: event-app-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: event-app
  template:
    metadata:
      labels:
        app: event-app
    spec:
      containers:
      - name: event-app
        image: event-app:1.0
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
```

```
metadata:
  name: event-app-service
spec:
  selector:
    app: event-app
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
  type: NodePort
```

Step 5 — Deploy using Kubernetes

Apply your manifest:

```
kubectl apply -f k8s.yaml
```

Kubernetes will:

1. Create a **Deployment** that manages two Pods (replicas)
2. Each Pod runs your **Docker container**
3. Create a **Service** to expose them on a stable IP/port

Step 6 — Verify everything

See your pods:

```
kubectl get pods
```

See your service:

```
kubectl get svc
```

You'll see something like:

```
event-app-service   NodePort   10.98.215.44   <none>
80:32244/TCP   1m
```

Step 7 — Access your app

If you're using **Minikube**:

```
minikube service event-app-service
```

If you're using **Docker Desktop**:
Open your browser:

`http://localhost:<NodePort>`
(e.g. http://localhost:32244)

✅ You'll see your `event_registration.html` file served from Kubernetes.

📦 5. How this represents Docker + Kubernetes integration

Tool	Role	Description
Docker	Builds the image	Packages your HTML and Nginx together
Kubernetes	Runs the image	Schedules, scales, and exposes the container
Image registry (optional)	Stores the image	Shared location for Kubernetes to pull from
`kubectl`	CLI control	Tells Kubernetes what to deploy and how

So Docker handles **build & package**, and Kubernetes handles **deployment & orchestration**.

🧠 6. Summary — What integration looks like conceptually

```
[Dockerfile + HTML]
        ↓
   docker build
        ↓
   event-app:1.0 (image)
        ↓
   kubectl apply -f k8s.yaml
        ↓
[Kubernetes]
  ↳ Schedules Pods running that Docker image
  ↳ Maintains 2 replicas (via Deployment)
  ↳ Exposes them via NodePort (via Service)
```

That's real-world **Docker–Kubernetes integration** in action.
You built once with Docker, then deployed anywhere with Kubernetes.

Now let's look at **how to stop (or clean up)** everything that Kubernetes is running for your app.
There are a few ways, depending on whether you want to **pause**, **scale down**, or **delete** completely.

🛑 OPTION 1: Completely stop & delete everything

If you just want to **stop everything you created from `k8s.yaml`**, run:

```
kubectl delete -f k8s.yaml
```

✅ What this does:

- Deletes the **Deployment** (which deletes all Pods it created)
- Deletes the **Service** (so the port is freed)
- Kubernetes automatically stops scheduling any more Pods for that app

You can confirm it's gone:

```
kubectl get all
```

You should see **no `event-app`** resources listed.

⚙️ OPTION 2: Keep configuration, but scale down (stop running Pods)

If you just want to **temporarily stop the app** but keep the Deployment definition in the cluster (so you can start it again later), you can scale it to zero replicas:

```
kubectl scale deployment event-app-deployment --replicas=0
```

✅ This means:

- The Deployment remains in the cluster.
- **All Pods are stopped and deleted**, so nothing runs.
- You can later restart it easily:

```
kubectl scale deployment event-app-deployment --
replicas=2
```

-

Check your pods afterward:

```
kubectl get pods
```
→ You should see **no running Pods**.

🧹 OPTION 3: Delete everything (Deployment, Service, image) + cleanup Docker

If you want to remove *everything* including your local image (to start clean):

```
kubectl delete -f k8s.yaml          # remove from cluster
docker rmi event-app:1.0            # remove image from local Docker
```
If using **Minikube**, and you loaded the image into its internal Docker:

```
minikube image rm event-app:1.0
```

III. Automate the process of running containerized application for exercise 7 using Kubernetes.

🧠 GOAL (in simple terms)

You already have:

- A **Dockerized app** (`event_registration.html` + `Dockerfile`)
- A **Kubernetes deployment file** (`k8s.yaml`)

Now you want to **automate the entire workflow**, so that every time you:

1. Change your code,
2. The container gets rebuilt,
3. And Kubernetes redeploys the new version — automatically.

This removes manual commands like:

```
docker build ...
kubectl apply ...
```

🧩 There are 3 levels of automation possible

Let's understand them briefly before we pick one:

Level	Tool	What it automates	Complexity
Basic	Shell script (e.g., `deploy.sh`)	Automates build + deploy commands	🟢 Easy
Intermediate	Makefile or Skaffold	Watches for changes, rebuilds + redeploys	🟡 Medium
Advanced	CI/CD (GitHub Actions / Jenkins / ArgoCD)	Full pipeline triggered by commits	🔴 High

We'll start with the **basic automation (bash script)** — the easiest and most common for learning.

📦 STEP 1 — Your setup so far

You already have:

```
event-app/
├── event_registration.html
├── Dockerfile
└── k8s.yaml
```

🧾 STEP 2 — Create a shell script to automate everything

Create a new file in the same directory called `deploy.sh`:

```
#!/bin/bash

# Build Docker image locally
docker build -t event-app:1.0 .

# Load the image into Minikube
minikube image load event-app:1.0

# Deploy to Kubernetes
kubectl apply -f k8s.yaml

# Show pods and services
kubectl get pods
kubectl get svc

# Open the service in the browser
minikube service event-app-service
```

🧩 STEP 3 — Make the script executable

Run:

```
chmod +x deploy.sh
```

Here :

Part	Meaning
`chmod`	"Change mode" → command used to modify file permissions
`+x`	Adds the “execute” permission to the file
`deploy.sh`	The file we’re giving permission to

So, `chmod +x deploy.sh` means:

“Give everyone permission to execute the file `deploy.sh` like a program.”

Now you can execute it with:

```
./deploy.sh   —— on MacOS

bash deploy.sh —— on Windows
```

🧰 STEP 4 — What this script does (line-by-line)

Line	Explanation
`#!/bin/bash`	Declares this is a Bash script
`docker build -t event-`	Builds the Docker image
`minikube image load event-`	If using Minikube, loads image inside its cluster
`kubectl apply -f k8s.yaml`	Deploys or updates your app in Kubernetes
`kubectl wait --for=condition=ready pod`	Waits until all Pods are ready
`minikube service event-app-`	Opens the running service in your default browser
`kubectl get svc`	If not using Minikube, shows the Service info so you can open it manually

💡 STEP 5 — Run and verify

Now simply execute:

```
./deploy.sh
```

You'll see logs like:

```
📦 Building Docker image...
🚀 Loading image into Minikube...
☸️  Deploying to Kubernetes...
⏳ Waiting for Pods to be ready...
pod/event-app-deployment-7f5b5b7df6-tjw8k condition met
🌐 Opening service in browser...
```

Then your browser will open automatically to your app 🎉

🧩 STEP 6 — Automating rebuilds (optional advanced feature)

If you want **continuous automatic redeployment** when files change (for example when you edit `event_registration.html`), you can use **Skaffold**.

🛠️ Install Skaffold

```
brew install skaffold
```

🧾 Create `skaffold.yaml`

```
apiVersion: skaffold/v4beta6
kind: Config
metadata:
  name: event-app
build:
  artifacts:
  - image: event-app
    context: .
deploy:
  kubectl:
```

```
    manifests:
      - k8s.yaml
```

🚀 Run Skaffold in dev mode

```
skaffold dev
```

✅ Skaffold will:

- Watch your local files
- Rebuild the Docker image if something changes
- Re-deploy automatically to Kubernetes

That's **true automation** of the container lifecycle.

🧭 STEP 7 — (Optional) Full CI/CD Automation

If you push your code to GitHub, you can add **GitHub Actions** to automatically:

1. Build the Docker image
2. Push it to Docker Hub
3. Deploy it to Kubernetes (via `kubectl`)

A simplified `.github/workflows/deploy.yml` could look like:

```
name: Deploy to Kubernetes

on:
  push:
    branches:
      - main

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Set up Docker
```

```
        uses: docker/setup-buildx-action@v3

      - name: Build and push image
        run: |
          docker build -t $DOCKER_USERNAME/event-app:latest .
          echo $DOCKER_PASSWORD | docker login -u
$DOCKER_USERNAME --password-stdin
          docker push $DOCKER_USERNAME/event-app:latest

      - name: Deploy to Kubernetes
        run: |
          kubectl apply -f k8s.yaml
```

This is **enterprise-level automation**, running on each push to GitHub.

✅ Summary

Step	Tool	Purpose
1	`Docker`	Build your container image
2	`Kubernetes`	Deploy and manage it
3	`deploy.sh`	Automate build + deploy manually
4	`Skaffold`	Auto rebuild/redeploy when code changes
5	`GitHub Actions`	CI/CD pipeline automation (on push)

🚀 Final Command to Remember

To fully automate and redeploy in one go (manually):

```
./deploy.sh
```