DevOps Full Course 2025 - Part 3
Docker Networking, Nginx Reverse Proxy, and Kubernetes (Ingress, Config, Storage)

Author: Sangam Mukherjee

Date: September 29, 2025

How To Use This Book

Who This Is For
This ebook is for engineers moving a MERN demo from Docker to Kubernetes on Windows (Minikube). It is written to be used during recording: every section ends with a hands-on checklist and copy-paste commands.

How To Read
1) Chapters 1 and 2 refresh the network and reverse proxy mental models that make everything else "click".
2) Chapters 3 to 6 build the exact Kubernetes foundation you will use in the demo.
3) Chapters 7 to 10 put a clean domain in front of multiple services with Ingress and show common pitfalls.
4) Chapter 11 adds persistence for Mongo so you do not lose data when pods restart.
5) The Appendices contain YAML and commands so you can reset and demo from scratch quickly.

1. Docker Networking - Mental Models and Common Pitfalls

1. Docker Networking - Mental Models and Common Pitfalls

1.1 Localhost Means "This Place"
- On your host: http://localhost refers to the host OS.
- Inside a container: http://localhost refers to that container only.
- Implication: from the frontend container, "localhost:5000" points to the frontend container, not the backend.

1.2 Bridges and DNS in Docker
- Default bridge: basic NAT, no automatic DNS for container names.
- User-defined bridge (recommended): has an embedded DNS so containers can call each other by name.
- Example: name your network "mern-net" and then call "backend:5000" and "mongo:27017" from the frontend and backend.

1.3 Correct Flow vs Broken Flow
- Wrong: Frontend (5173) -> localhost:5000 (points to itself)
- Right: Frontend (5173) -> backend:5000 -> mongo:27017 (resolves via Docker DNS)

1.4 Publishing Ports vs Internal Ports
- "ports: 5000:5000" exposes a container port to the host.
- Inside the user-defined bridge, containers talk directly without publishing.

1.5 Quick Lab
- Create a user-defined bridge: docker network create mern-net
- Run mongo and backend on mern-net.

- From backend container, ping mongo and connect via mongo:27017.
- From host, curl http://localhost:5000/health only if you published the backend port.

1.6 Troubleshooting
- "Could not resolve host": ensure services are attached to the same user-defined network.
- "ECONNREFUSED": the target container is not listening on that port, the port is not published to the host, or the service name is wrong.

2. Nginx Reverse Proxy - Why and How

2. Nginx Reverse Proxy - Why and How (MERN Example)

2.1 Why Use a Reverse Proxy
- One clean entrypoint (http://localhost on your host, or a single domain in k8s).
- Path based routing: "/" to frontend, "/api" to backend.
- Central place for gzip, caching, logs, rate limits, and TLS.

2.2 Docker Compose Pattern (Optional Pre-K8s)
- Run the frontend and backend, then add an Nginx container that forwards:
- "/" -> frontend:5173
- "/api" -> backend:5000

2.3 Minimal nginx.conf (Docker Demo)

```
      events {}
      http {
        upstream ui { server frontend:5173; }
        upstream api { server backend:5000; }
        server {
          listen 80;
          location / { proxy_pass http://ui; }
          location /api/ { proxy_pass http://api/; }
}
}
```

2.4 Recording Tips
- Always show why the proxy exists before showing how to configure it.
- Keep the config small so the mental model stays clear.

3. Build-time ARG vs Runtime ENV (Vite)

3. Build-time ARG vs Runtime ENV for Vite

3.1 The Rule
- Vite replaces "import.meta.env.VITE_*" at build time. The built assets will not change if you

change env after the build.

3.2 What To Do
- Pass the API base while building the frontend image.
- Example: docker build -t frontend:local -f client/Dockerfile --build-arg VITE_API_URL=/api client

3.3 Dockerfile Snippet
ARG VITE_API_URL=/api
ENV VITE_API_URL=$VITE_API_URL
RUN npm run build

3.4 Common Mistakes
- Expecting a new container ENV to change the already-built JS. It will not.
- Mixing "npm run preview" (Dev server) with production static build behavior.

4. Kubernetes Fundamentals - Declarative Beats Imperative

4. Kubernetes Fundamentals - Declarative Beats Imperative

4.1 Two Styles
- Imperative: "kubectl run", "kubectl set image". Quick but hard to repeat.
- Declarative: write YAML that describes the desired state and "kubectl apply -f". Repeatable, versionable, and scales well.

4.2 Core Objects You Will Use
- Pod: one or more containers that share the IP and volumes.
- ReplicaSet: keeps N replicas of a pod.
- Deployment: manages ReplicaSets and rolling updates/rollbacks.
- Service: stable virtual IP that load-balances to matching pods.
- ConfigMap and Secret: externalize non-secret and secret configuration.

4.3 Services 101
- ClusterIP (default): in-cluster only.
- NodePort: exposes a high port on each node (clunky for local demos).
- LoadBalancer: cloud-friendly. In minikube, use Ingress instead.

5. Namespaces, Labels, and Service Selection

5. Namespaces, Labels, and Service Selection

5.1 Namespaces
- Use a namespace per demo: "devops-part3". Keeps resources grouped and easy to delete.

5.2 Labels and Selectors

- Label your pods: "app: backend".
- Service selector uses the same label to target pods. If labels mismatch, the Service has no endpoints.

5.3 Quick Checks
- kubectl -n devops-part3 get pods -o wide
- kubectl -n devops-part3 get svc
- kubectl -n devops-part3 get endpoints backend

6. From Docker Compose to Kubernetes - Step by Step

6. From Docker Compose to Kubernetes - Step by Step

6.1 Folder
- Create "k8s/" with files: 00-namespace.yaml, 01-mongo.yaml, 02-backend.yaml, 03-frontend.yaml, 04-ingress.yaml, 05-mongo-pvc.yaml.

6.2 Namespace

```
apiVersion: v1
kind: Namespace
metadata:
```

name: devops-part3

6.3 Mongo (Deployment + Service)
- 1 replica, image "mongo:6.0". For persistence, mount a volume at "/data/db".
- Service "mongo" so other pods can resolve "mongo:27017".

6.4 Backend (Deployment + Service)
- 2 replicas. Env MONGO_URI=mongodb://mongo:27017/taskdb.
- Expose port 5000. Add simple /health for readiness.
- Service "backend" to map port 5000 inside the cluster.

6.5 Frontend (Deployment + Service)
- Build with VITE_API_URL=/api so the browser calls "/api" on the same origin.
- Service "frontend" to map port 5173.

6.6 Apply In Order
- kubectl apply -f k8s/00-namespace.yaml
- kubectl apply -f k8s/01-mongo.yaml
- kubectl apply -f k8s/02-backend.yaml
- kubectl apply -f k8s/03-frontend.yaml
- Verify pods and services before moving to Ingress.

7. Config and Secrets - 12 Factor in Kubernetes

7. Config and Secrets - 12 Factor in Kubernetes

7.1 ConfigMap

```
kind: ConfigMap
apiVersion: v1
metadata:
```

name: app-config
namespace: devops-part3
data:
FRONTEND_BASE: "/"
MONGO_HOST: "mongo"

7.2 Secret (Opaque)

```
kind: Secret
apiVersion: v1
metadata:
```

name: mongo-secret
namespace: devops-part3
type: Opaque
data:
MONGO_URI: bW9uZ29kYjovL21vbmdvOjI3MDE3L3Rhc2tkYg==

7.3 Consume in Deployment
envFrom:
- configMapRef:

```
name: app-config
```

- secretRef:

```
name: mongo-secret
```

7.4 Rolling Out Changes
- kubectl -n devops-part3 rollout restart deploy/backend
- kubectl -n devops-part3 logs -l app=backend --tail=100

8. Ingress - One Domain, Many Services

8. Ingress - One Domain, Many Services

8.1 Why Ingress
- NodePort gives odd ports per service and is hard to remember.
- Ingress places a smart HTTP router at the edge (nginx controller in minikube) so you can map a real host to multiple services.

8.2 Anatomy of a Rule
- Host: merndemo.local
- Path: "/" to frontend Service port 5173
- Path: "/api" to backend Service port 5000
- Optional rewrite-target "/" for consistent path handling.

8.3 Windows Hosts Mapping
- Edit: C:\Windows\System32\drivers\etc\hosts
- Add: 127.0.0.1 merndemo.local

9. Minikube Ingress Walkthrough

9. Minikube Ingress Walkthrough

9.1 Enable Addon

```
minikube addons enable ingress
```

9.2 Verify Controller

```
kubectl get pods -n ingress-nginx
kubectl get svc  -n ingress-nginx
```

9.3 Apply Ingress Manifest

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
```

name: mern-ingress
namespace: devops-part3
annotations:

```
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
    - host: merndemo.local
      http:
        paths:
          - path: /api
            pathType: Prefix
            backend:
              service:
                name: backend
                port: { number: 5000 }
          - path: /
            pathType: Prefix
            backend:
              service:
                name: frontend
                port: { number: 5173 }
```

9.4 Tunnel
- Run: minikube tunnel (keep this terminal open).

9.5 Test

```
curl http://merndemo.local/
curl http://merndemo.local/api/tasks
```

10. Frontend Build for Ingress - Avoid CORS

10. Frontend Build for Ingress - Avoid CORS

10.1 Why This Works
- The browser calls "/api" on the same origin "merndemo.local". Ingress routes "/api" to backend.
No cross origin, no CORS preflight.

10.2 Rebuild and Reload

```
docker build -t frontend:local -f client/Dockerfile --build-arg VITE_API_URL=/api client
minikube image load frontend:local
kubectl -n devops-part3 rollout restart deploy/frontend
```

11. Storage and Persistence - PVC for Mongo

11. Storage and Persistence - PVC for Mongo

11.1 Problem With emptyDir
- Data vanishes when pods restart or reschedule.

11.2 PVC Definition

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
```

name: mongo-pvc
namespace: devops-part3

```
spec:
  accessModes: [ "ReadWriteOnce" ]
  resources:
    requests:
      storage: 1Gi
```

11.3 Mount in Deployment

```
volumes:
```

- name: mongo-data

```
    persistentVolumeClaim:
      claimName: mongo-pvc
```

```
      volumeMounts:
```

- name: mongo-data

```
          mountPath: /data/db
```

11.4 Verify

```
      kubectl -n devops-part3 get pvc
      kubectl -n devops-part3 describe pvc mongo-pvc
```

12. Troubleshooting Annex

12. Troubleshooting Annex

12.1 Minikube Ingress "error validating ... failed to download openapi ... connect: connection refused"
- Cause: API server not healthy yet when addon yaml applies.
- Fix A: rerun "minikube addons enable ingress" after "minikube status" shows Running.
- Fix B: retry "kubectl apply -f" with "--validate=false" only if you must, then check "kubectl get pods -n ingress-nginx".
- Fix C: "minikube stop" then "minikube start --driver=docker".

12.2 Patching Deployment Strategy on Windows PowerShell
- Correct quoting for JSON payload:

```
      kubectl -n devops-part3 patch deploy mongo --type merge -p '{"spec":{"strategy":{"type":"Recr
```

- Or use a file "patch.json" and run:

```
      kubectl -n devops-part3 patch deploy mongo --type merge --patch-file=patch.json
```

12.3 No Endpoints on Service
- Check labels on pod template vs selector in Service.
- Example: "app: backend" must match in both places.

12.4 Frontend Cannot Reach Backend
- If calling "http://localhost:5000" from the browser while using Ingress, change to "/api".
- If calling from inside frontend container, do not use "localhost"; use "backend:5000".

12.5 DNS and Hosts
- After editing hosts on Windows, open a new terminal or flush DNS: "ipconfig /flushdns".

Appendix A - YAML Snippets

Appendix A - YAML Snippets (Copy Ready)

00-namespace.yaml

```
      ----------------
```

```
apiVersion: v1
kind: Namespace
metadata:
```

name: devops-part3

01-mongo.yaml (Deployment and Service)

```
---------------------------------------
apiVersion: apps/v1
kind: Deployment
metadata:
```

name: mongo
namespace: devops-part3
labels:

```
    app: mongo
spec:
```

replicas: 1
selector:

```
    matchLabels:
      app: mongo
```

template:

```
    metadata:
      labels:
        app: mongo
    spec:
      containers:
        - name: mongo
          image: mongo:6.0
          ports:
            - containerPort: 27017
          volumeMounts:
            - name: mongo-data
              mountPath: /data/db
      volumes:
        - name: mongo-data
          emptyDir: {}

---
apiVersion: v1
kind: Service
metadata:
```

name: mongo
namespace: devops-part3

```
spec:
```

selector:

```
    app: mongo
  ports:
    - port: 27017
```

```
    targetPort: 27017
```

02-backend.yaml (Deployment and Service)

```
---------------------------------------
apiVersion: apps/v1
kind: Deployment
metadata:
```

name: backend

namespace: devops-part3

labels:

```
    app: backend
spec:
```

replicas: 2

selector:

```
    matchLabels:
      app: backend
```

template:

```
    metadata:
      labels:
        app: backend
    spec:
      containers:
        - name: backend
          image: backend:local
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 5000
          env:
            - name: MONGO_URI
              value: mongodb://mongo:27017/taskdb

---
apiVersion: v1
kind: Service
metadata:
```

name: backend

namespace: devops-part3

```
spec:
```

selector:

```
    app: backend
  ports:
    - port: 5000
      targetPort: 5000
```

03-frontend.yaml (Deployment and Service)

```
---------------------------------------
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
```

name: frontend
namespace: devops-part3
labels:

```
    app: frontend
spec:
```

replicas: 1
selector:

```
    matchLabels:
      app: frontend
```

template:

```
    metadata:
      labels:
        app: frontend
    spec:
      containers:
        - name: frontend
          image: frontend:local
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 5173

---
apiVersion: v1
kind: Service
metadata:
```

name: frontend
namespace: devops-part3

```
spec:
```

selector:

```
    app: frontend
  ports:
    - port: 5173
      targetPort: 5173
```

04-ingress.yaml

```
---------------
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
```

name: mern-ingress
namespace: devops-part3
annotations:

```
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
```

```
  rules:
    - host: merndemo.local
      http:
        paths:
          - path: /api
            pathType: Prefix
            backend:
              service:
                name: backend
                port:
                  number: 5000
          - path: /
            pathType: Prefix
            backend:
              service:
                name: frontend
                port:
                  number: 5173
```

05-mongo-pvc.yaml

```
-----------------
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
```

name: mongo-pvc

namespace: devops-part3

```
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Appendix B - Command Sequences

Appendix B - Command Sequences

B.1 Clean Up

```
kubectl delete -f k8s/04-ingress.yaml --ignore-not-found
kubectl delete -f k8s/03-frontend.yaml --ignore-not-found
kubectl delete -f k8s/02-backend.yaml  --ignore-not-found
kubectl delete -f k8s/01-mongo.yaml    --ignore-not-found
kubectl delete -f k8s/05-mongo-pvc.yaml --ignore-not-found
kubectl delete -f k8s/00-namespace.yaml --ignore-not-found
kubectl delete namespace devops-part3 --ignore-not-found
kubectl -n devops-part3 delete pvc --all --ignore-not-found
```

B.2 Optional Image Clean

```
minikube image rm backend:local  2>$null
minikube image rm frontend:local 2>$null
docker rmi backend:local frontend:local 2>$null
```

B.3 Build

```
docker build -t backend:local  -f server/Dockerfile server/
docker build -t frontend:local -f client/Dockerfile --build-arg VITE_API_URL=/api client/
```

B.4 Load Into Minikube

```
minikube image load backend:local
minikube image load frontend:local
```

B.5 Apply

```
kubectl apply -f k8s/00-namespace.yaml
kubectl apply -f k8s/01-mongo.yaml
kubectl -n devops-part3 wait --for=condition=ready pod -l app=mongo --timeout=180s
kubectl apply -f k8s/02-backend.yaml
kubectl -n devops-part3 wait --for=condition=ready pod -l app=backend --timeout=180s
kubectl apply -f k8s/03-frontend.yaml
kubectl -n devops-part3 wait --for=condition=ready pod -l app=frontend --timeout=180s
minikube addons enable ingress
kubectl apply -f k8s/04-ingress.yaml
minikube tunnel  # keep this open
```

B.6 Test

```
curl -v http://merndemo.local/
curl -v http://merndemo.local/api/tasks
kubectl -n devops-part3 logs -l app=backend --tail=200
```

Appendix C - Your Original Notes (Cleaned)

Appendix C - Your Original Notes (Cleaned to ASCII)
Below is a verbatim dump of the notes file you uploaded, cleaned to ASCII so there are no missing glyphs. Use this if you want to cross-check exact phrasing while recording.

```
------------------------------------------------------------
```

Docker Networking & Nginx
A container is an isolated environment: own filesystem, own processes, own networking stack.
Analogy: Think of containers as apartments in the same building. Each has its own kitchen, bathroom, and mailbox.
By default, one apartment's mailbox doesn't receive letters meant for another. Same way: one container doesn't "see" another's processes or ports unless there's a.
Localhost Misunderstanding (the #1 beginner mistake)
Outside containers: localhost means your host machine.
Inside container: localhost means this container only.
That's why:
From host: curl localhost:5000 -> hits backend (because you published port in compose).
From frontend container: curl localhost:5000 -> fails (because no backend process inside frontend).
Docker Networking Basics
Bridge network (default): Containers get isolated private IPs. Without explicit config, they can't talk to each other by name.
User-defined bridge (like mern-net):

Docker runs an embedded DNS server.
Containers can talk to each other by service name (e.g., backend:5000).
This is like giving every apartment in the building a nameplate outside its door, instead of forcing people to memorize IP addresses.
Advantages of Container-to-Container Networking
Service Discovery: IPs change every run. Names don't. (Exactly what Kubernetes DNS does later.)
Security: Mongo is never exposed to outside world -> reduces attack surface.
Port collisions avoided: Two different containers can both use port 5000 internally - no clash, because they're isolated.
Scalability: If you add multiple backends, they still sit behind backend:5000.
Diagrammatic Flow
Wrong (using localhost):
Frontend container (5173) ---> localhost:5000 (points to itself [x])
Correct (using Docker DNS):
Frontend (5173) ---> backend:5000 ---> mongo:27017
With Nginx:
Browser ---> Nginx (80)

```
                    |--- /api ---> backend:5000
                    |--- /      ---> frontend:5173
```

Nginx Reverse Proxy
What is Nginx?
High-performance web server + reverse proxy.
Handles millions of concurrent connections -> that's why 30%+ of websites use it.
Role as reverse proxy: Client thinks it's talking to one server, but Nginx quietly routes traffic to the correct service.
Why Nginx in containerized setups?
To hide multiple services behind one clean entrypoint.
To implement path-based routing (/api vs /).
To centralize concerns: SSL termination, compression, caching, logging.
To prepare students for Kubernetes Ingress, which uses Nginx or Traefik under the hood.
3. Reverse Proxy vs Forward Proxy
Forward proxy: Client -> Proxy -> Internet (e.g., corporate proxy, VPN).
Reverse proxy: Client -> Proxy -> Private services (what we need).
Reverse proxy protects and simplifies access.
4. Real-World Use
Netflix uses Envoy (like Nginx) for service-to-service communication.
GitHub serves static files with Nginx at the edge.
Almost every cloud-native deployment uses a reverse proxy in front of microservices.
5. Flow with Nginx in MERN
Without Nginx:
Frontend -> http://localhost:5173
Backend -> http://localhost:5000

With Nginx:
Both behind http://localhost (clean).
/ goes to frontend, /api goes to backend.

Applying in MERN (Detailed Walkthrough)
nginx.conf

```
docker compose file change
```

explanation
4) Why set VITE_API_URL as ARG vs ENV - deep explanation (no nonsense)
You asked: "why we need to add frontend env as we have arg inside docker file?" - here's the precise behavior and recommended practice.
Build-time ARG (what it is)
ARG is available only at build time in the Dockerfile.
Example: ARG VITE_API_URL -> you can pass value when building (docker build --build-arg VITE_API_URL=/api).
Use case: values needed by build tools (e.g., Vite) to produce a static bundle.
ENV in Dockerfile (what it is)
ENV sets environment variables in the resulting image and are available at runtime to processes started inside the container.
But - if your frontend is a static build (React/Vite npm run build produces HTML/CSS/JS), the JS files are already baked with the build-time value. Changing ENV at runtime typically does not alter the built JS content.
Vite & VITE_ variables specifics
Vite injects import.meta.env.VITE_* at build time into the bundle. So:
If you want the JS files to reference VITE_API_URL, you must provide it when building.
Common pattern:
Build-time ARG VITE_API_URL -> set ENV VITE_API_URL=$VITE_API_URL -> npm run build reads it and embeds into static files.
That's what your client Dockerfile does when you pass ARG and set ENV before npm run build.
Why pass VITE_API_URL in docker-compose build args?
Because nginx is the external entrypoint and the client will be served by nginx under /, the cleanest client code is to call relative paths, e.g., VITE_API_URL = /api (or better: code uses relative fetch('/api/tasks')).
Passing VITE_API_URL: "/api" as a build-arg makes your built JS use /api as the base. This is portable: whether served by nginx, deployed behind a load balancer, or in k8s ingress, /api will be proxied to the backend.
Summary (practical rules)
If frontend is a static build (production): set the API base at build time (ARG -> ENV -> build).
Changing the container runtime ENV after build won't change the built JS.
If frontend runs in dev mode (vite dev server) and expects runtime variables, it may read from .env or runtime env; but Vite dev server behavior differs: it reads .env files at startup.
For maximum simplicity in demos and production:
Build frontend with VITE_API_URL=/api.

Make client use relative /api requests (or import.meta.env.VITE_API_URL if you need explicit base).
Let nginx handle routing, so host URL is always http://localhost and API is http://localhost/api.
5) Practical verification steps (quick checklist you can run in class)
Build & run:

```
docker-compose up --build
```

Check containers:

```
docker ps
# expect: mongo, backend, frontend, nginx
```

Verify nginx serving frontend:
Open browser -> http://localhost/ -> React UI should load.
Verify API through nginx:
From browser app, create a task -> network tab shows request to http://localhost/api/tasks -> nginx forwarded it to backend -> backend saved in mongo.

Imperative vs Declarative & Core Declarative Objects
Why Declarative (deep, teachable)
Start here in class: declarative is not just a YAML file - it's a whole workflow.
Core idea:
Imperative = tell the system how to do something step-by-step (kubectl create pod ..., kubectl set image ...).
Declarative = describe the desired state and let the system converge to it (store that desired state in YAML/JSON, run kubectl apply -f). Kubernetes controllers operate continuously to reconcile actual state -> desired state.
Why declarative is standard - deep reasons to explain:
Idempotence: Re-applying the same manifest results in the same cluster state - no unexpected side effects.
Auditability & Versioning: YAML in Git = infra-as-code. You can review, PR, revert, and trace who changed what.
Automation/GitOps: Tools (ArgoCD/Flux) watch git and reconcile automatically - declarative makes automation safe.
Drift detection & self-healing: Controllers continuously reconcile drift (e.g., if node kills a pod, ReplicaSet recreates).
Testing & previewing: You can lint, dry-run, and validate manifests; you can promote the same manifest across envs.
Scalability of ops teams: Hundreds of clusters/environments manageably controlled through declarative code.

Core Objects
2.1 Pod (concept)
Smallest deployable unit. One or more containers share:

network namespace (same IP) -> containers inside Pod can talk to each other via localhost.
volumes (shared filesystem).
Pods are ephemeral. If a Pod dies, unless a controller recreates it, it may remain terminated.
Use case: debugging, sidecar patterns (logging, proxy), short-run jobs (init containers).

2.2 ReplicaSet (concept)
Ensures N replicas of a Pod run. If a Pod crashes, ReplicaSet creates a new one.
Uses label selectors to manage Pods.
Usually auto-created/managed by Deployments - you rarely hand-write ReplicaSets in production.
Diagram
ReplicaSet backend-rs (replicas: 3)
Pod backend-1
Pod backend-2
Pod backend-3
2.3 Deployment (concept)
Higher-level controller that manages ReplicaSets and provides:
Rolling Updates (zero-downtime if configured)
Rollbacks
Revision history
Deployment backend
ReplicaSet backend-v1
pods...
ReplicaSet backend-v2 (new)

```
pods...
```

2.4 Service (concept)
Stable network endpoint (DNS + clusterIP) that forwards to Pods.
Types:
ClusterIP (default) - internal only
NodePort - exposes on all node IPs (high port range) for simple external reachability
LoadBalancer - cloud-managed external LB
Headless (clusterIP: None) - for stateful services or direct pod discovery
Diagram
Client(frontend) --> Service (backend) --> kube-proxy -> pod endpoints
Resource Management
1. Namespaces
Think of namespaces like folders in your cluster.
They separate resources so you don't mix dev, staging, prod.
We've already been using devops-part3.
Why?
Avoid naming conflicts (backend in dev vs backend in prod).
Apply policies or quotas per namespace.
Easier cleanup (kubectl delete ns devops-part3 removes all demo resources).
2. Labels & Selectors

Labels are simple key-value tags you put on resources.
Selectors are queries that use labels to find resources.
Services don't know pod IPs (they change). They use labels to know which pods belong.
Example in your backend Deployment:
labels:
app: backend
And in backend Service:
selector:
app: backend
This is how the Service knows which pods to send traffic to.

Files: complete, annotated, ready to drop in k8s/

```
minikube start --driver=docker
```

Create a folder k8s/ at the repo root and add these files (in this order). I numbered files so you can kubectl apply -f k8s/ in one shot
create k8s/00-namespace.yaml -> desktop

```
apiVersion: v1
```

Which API group/version this resource uses. Namespaces live in the core v1 API. (Every Kubernetes object needs an apiVersion.)

```
kind: Namespace
```

The resource type. This instructs Kubernetes to create a namespace object.

```
metadata:
```

Metadata block for identifying the object.
name: devops-part3
The human-readable name assigned to this namespace. All subsequent demo resources will be created inside this namespace so they don't pollute the cluster and can be cleaned up together.

2. create k8s/01-mongo.yaml (Deployment + Service) -> desktop
Line-by-line explanation - Deployment part

```
# Deployment of a single mongo pod (demo)
```

Comment; not processed by k8s. Helpful for humans.

```
apiVersion: apps/v1
```

Deployments are in the apps/v1 API group/version.

```
kind: Deployment
```

Create a Deployment (controller that manages ReplicaSets and Pods).

```
metadata:
```

Object identification info follows.
name: mongo
Resource name for this Deployment (mongo).
namespace: devops-part3
The namespace we created earlier; ensures this Deployment is scoped to the demo namespace.
labels:
Key/value pairs to tag/identify this resource (useful for selectors and human filtering).
app: mongo
A label indicating this resource belongs to the mongo app group.

```
spec:
```

The desired state spec for the Deployment.
replicas: 1
Desired number of Pod replicas (we want one Mongo pod in this demo).
selector:
How the Deployment identifies which Pods it manages.
matchLabels:
The selector uses labels; it will adopt pods matching these labels.
app: mongo
Selector matches pods labeled app=mongo.
template:
Pod template: the spec used to create Pods.
template.metadata.labels:
Labels applied to Pods created by this Deployment (must match selector).

```
spec: (inside template)
```

Pod spec - containers and volumes.

```
containers:
```

List of containers inside the Pod. Here we have one container for MongoDB.
- name: mongo
Container name inside the Pod.
image: mongo:6.0
Container image to run (from Docker Hub). Version 6.0 is used for the demo.

```
ports:
```

Declares container ports - documents internal listening port.
- containerPort: 27017
Mongo's default port; this is purely metadata for Kubernetes & readers (and used by some tools).

```
volumeMounts:
```

Mounts volumes into the container filesystem.
- name: mongo-data
Name of the volume (must match a volumes: entry).
mountPath: /data/db

Where in the container the volume is mounted (Mongo stores DB files here).

```
volumes:
```

Pod-level volume definitions.
- name: mongo-data
Volume named mongo-data.
emptyDir: {}
A simple ephemeral volume allocated on the node; data is lost when Pod is removed. Used here to avoid PVC complexity for teaching.
Why these choices: single mongo pod simplifies demo; emptyDir avoids teaching PVC/storage setup yet allows Mongo to write files.
Line-by-line explanation - Service part

```
---
```

YAML document separator - now a new resource in same file.

```
apiVersion: v1
```

Services are part of core v1.

```
kind: Service
```

Create a Service object.

```
metadata:
```

Resource identification.
name: mongo
The Service name; this is the DNS name pods will use to reach the DB inside the namespace.
namespace: devops-part3
Same namespace as the Deployment.

```
spec:
```

Service specification follows.
selector:
Selects pods to route traffic to.
app: mongo
Picks pods labeled app=mongo (our mongo pod).

```
ports:
```

Port mapping for the Service.
- port: 27017
Service port (the port other pods use when they talk to the service).
targetPort: 27017
Pod container port to forward traffic to (our mongo pod's containerPort).
type: ClusterIP
The service is internal-only - exposes a stable virtual IP inside the cluster. This is exactly what we

want for DB.
Why Service: gives a stable hostname mongo.devops-part3.svc.cluster.local (or simply mongo inside the namespace) that the backend can use - pod IPs change but service name stays stable.

3. create k8s/02-backend.yaml (Deployment + ClusterIP Service) -> desktop
Line-by-line explanation - Deployment part

```
# Backend deployment (app) + ClusterIP service
```

Human comment.

```
apiVersion: apps/v1 / kind: Deployment
```

Use Deployment controller.
metadata.name: backend
Resource name backend.
namespace: devops-part3
Scopes resource to demo namespace.
labels: app: backend
Label for resource grouping.
spec.replicas: 2
Desired pod replicas (we use 2 to demonstrate scaling/load balancing).
selector.matchLabels: app: backend
Identifies pods managed by the Deployment.
template.metadata.labels: app: backend
Labels applied to Pods (must match selector).

```
containers: (pod's containers)
```

- name: backend
Container's name.
image: backend:local
Image to run. Using backend:local implies you built & loaded this image into the local cluster. In production you'd use myregistry/mybackend:1.2.3.
imagePullPolicy: IfNotPresent
Kubelet will pull the image only if it's not present on the node; useful for local dev images.

```
ports: - containerPort: 5000
```

Declares the container's listening port (backend server's port).

```
env:
```

Environment variables injected into the container.
- name: MONGO_URI / value: "mongodb://mongo:27017/taskdb"
The backend reads this env var to connect to Mongo. It uses the Service DNS name mongo which resolves inside the namespace to the mongo Service.
readinessProbe:
Probe to determine whether the container is ready to receive traffic. Only ready pods are added to the Service endpoints.

httpGet: path: /health port: 5000 - performs an HTTP GET to /health.
initialDelaySeconds: 3 - wait 3s after container start before probing.
periodSeconds: 5 - probe every 5s.
failureThreshold: 3 - 3 consecutive failures mark it not ready.
livenessProbe:
Probe to tell kubelet whether container is alive.
Similar config points; if failing, kubelet restarts the container.
initialDelaySeconds: 10 - give the app more time before starting liveness checks (so startup isn't mistaken as failure).
Why these choices: readiness prevents traffic to pods that haven't connected to DB yet; liveness enables automatic restart if the process deadlocks.
Line-by-line explanation - Service part

```
---
```

Document separator.

```
apiVersion: v1 / kind: Service
```

Define a Service.
metadata.name: backend
Service name is backend.
namespace: devops-part3
Same namespace.
spec.selector: app: backend
Routes to pods labeled app=backend (the pods created by Deployment).

```
ports:
```

- protocol: TCP - protocol.
port: 5000 - service port that clients inside the cluster use.
targetPort: 5000 - container port which receives traffic.
type: ClusterIP
Internal-only service.
Why Service: frontend or other internal clients use http://backend:5000 and Kubernetes routes requests to one of the backend pods based on the service.

4. create 03-frontend.yaml (Deployment + ClusterIP Service) -> desktop
What this file does (teachable):
Runs the frontend container inside cluster.
You must build frontend:local with VITE_API_URL set to the in-cluster backend address (http://backend:5000/api) so the UI calls the backend by service name.

How these files work together (flow you can draw on board)
Namespace groups objects.
mongo Deployment launches a Mongo pod and mongo Service gives a stable DNS name.
DNS inside cluster resolves mongo -> cluster IP -> kube-proxy routes to mongo pod.

backend Deployment launches 2 backend pods. Each pod reads MONGO_URI env and connects to mongodb://mongo:27017/taskdb.
backend Service gives backend a stable internal endpoint; other pods call http://backend:5000.
Readiness probes ensure backend pods are only routed to after they pass /health.

run order

SECTION A - Cleanup existing cluster resources & images

```
# 1) Delete the demo namespace and everything inside it (quickest full cleanup).
#    This removes Deployments, Services, Pods, PVCs etc that live in the namespace.
kubectl delete namespace devops-part3

# 2) In case namespace deletion hangs or you previously created resources in default ns,
#    explicitly delete any leftover demo resources by label or by file (safe idempotent).
kubectl delete all --all -n devops-part3      # attempt to delete all in ns (may error if ns

# 3) Also delete PVCs (persisted volumes) in that namespace (if any)
kubectl -n devops-part3 delete pvc --all

# 4) If delete by namespace didn't fully succeed, force-delete specific manifests (safe to re
kubectl delete -f k8s/03-frontend.yaml --ignore-not-found
kubectl delete -f k8s/02-backend.yaml --ignore-not-found
kubectl delete -f k8s/01-mongo.yaml --ignore-not-found
kubectl delete -f k8s/00-namespace.yaml --ignore-not-found

# 5) Wait until the namespace is gone before proceeding. Poll until it's removed.
```

while (kubectl get namespace devops-part3 -o json -ErrorAction SilentlyContinue) {
Write-Host "Waiting for namespace devops-part3 to terminate..." -ForegroundColor Yellow
Start-Sleep -Seconds 3
}
Write-Host "Namespace devops-part3 removed (or did not exist)." -ForegroundColor Green

Remove images from minikube image store (if loaded previously).

```
#    These commands will clean minikube's local image store so re-loading is clean.
minikube image rm backend:local 2>$null
minikube image rm frontend:local 2>$null

# 7) Optionally remove host Docker images to free tags (safe; will fail harmlessly if missing
docker rmi backend:local frontend:local 2>$null
```

Build images locally

```
# 1) Build backend image on your host Docker
#    -t backend:local -> tag for k8s to reference
docker build -t backend:local -f server/Dockerfile server/

# 2) Build frontend image with VITE_API_URL pointing to localhost
#    This bakes the API base as http://localhost:5000/api into the built frontend static file
docker build --build-arg VITE_API_URL=http://localhost:5000/api -t frontend:local -f client/D
```

Load images into minikube

```
# Load the locally-built images into the minikube image store (no registry push)
minikube image load backend:local
minikube image load frontend:local

# Quick confirm that minikube has the images (optional)
minikube image ls | Select-String "backend"
minikube image ls | Select-String "frontend"
```

Apply k8s manifests in the correct order (namespace -> mongo -> backend -> frontend)

```
# 1) Create namespace
kubectl apply -f k8s/00-namespace.yaml

# 2) Apply MongoDB (Deployment + Service + PVC)
kubectl apply -f k8s/01-mongo.yaml

# 3) Wait for Mongo pod to be Ready (block until Ready or timeout)
kubectl -n devops-part3 wait --for=condition=ready pod -l app=mongo --timeout=180s
kubectl -n devops-part3 get pods -l app=mongo -o wide

# 4) Deploy backend (Deployment + Service). Must reference image: backend:local
kubectl apply -f k8s/02-backend.yaml

# 5) Wait for backend pods to be Ready
kubectl -n devops-part3 wait --for=condition=ready pod -l app=backend --timeout=180s
kubectl -n devops-part3 get pods -l app=backend -o wide
kubectl -n devops-part3 get svc backend -o wide

# 6) Deploy frontend (Deployment + Service). Must reference image: frontend:local
kubectl apply -f k8s/03-frontend.yaml

# 7) Wait for frontend pods to be Ready
kubectl -n devops-part3 wait --for=condition=ready pod -l app=frontend --timeout=180s
kubectl -n devops-part3 get pods -l app=frontend -o wide
kubectl -n devops-part3 get svc frontend -o wide
```

Final verification

```
# Show pods and services in the namespace with details
kubectl -n devops-part3 get pods,svc -o wide
```

Port-forward (two terminals)
Open Terminal A and run (keeps running - do not close if you want live forwarding):

```
# Terminal A: port-forward backend first so browser requests to http://localhost:5000 succeed
kubectl -n devops-part3 port-forward svc/backend 5000:5000
# Keep this terminal open.
```

Open Terminal B and run:

```
# Terminal B: port-forward frontend so you can open the UI at localhost:5173
kubectl -n devops-part3 port-forward svc/frontend 5173:5173
# Keep this terminal open.
```

Now open your browser:

Configurations

Why configurations matter in Kubernetes

Apps rarely work with just code. They also need configuration:
Non-sensitive: API URLs, feature flags, logging levels, DB hostnames.
Sensitive: passwords, API keys, DB URIs, certificates.
Hardcoding these inside images (like we did with MONGO_URI in backend Deployment) is bad because:
You need to rebuild images if config changes.
You risk exposing secrets in your Git repo or Dockerfile.
You cannot separate environments (dev/stage/prod).
Kubernetes gives two built-in objects to solve this:
ConfigMap
Stores non-sensitive config in key-value pairs.
Examples: MONGO_HOST=mongo, LOG_LEVEL=debug, FRONTEND_URL=/.
Can be consumed by Pods as:
Environment variables.
Mounted as files inside a volume.
Advantage: update configs without rebuilding images.
Limitation: not encrypted, visible to anyone with access.
Secret
Stores sensitive config like passwords, tokens, DB URIs.
Values are base64-encoded in YAML (not real encryption, but better than plain text).
Can be consumed the same way as ConfigMaps (env vars or files).
Can be integrated with external secret managers (Vault, AWS Secrets Manager, etc.).
Best practice: never check secrets YAML into public repos. Use kubectl create secret commands instead.

Flow (Config injection)
ConfigMap / Secret

Pod spec (Deployment)

via env:

```
MONGO_HOST
MONGO_URI
```

via volume:

```
mounted file
```

Configs live outside code.
Pods consume them dynamically.
One config can be shared across multiple apps.
Part 3 - YAML files
We'll add 3 files to k8s/:
k8s/configmap.yaml
Explanation:

data stores key-value pairs (all plain text).
MONGO_HOST will be combined with DB name by backend.
ConfigMap can hold multiple values - we keep it simple here.

k8s/secret-mongo.yaml
type: Opaque -> generic key-value secret.
data: values must be base64-encoded.
echo -n "mongodb://mongo:27017/taskdb" | base64 -> gives the encoded string.
This secret holds full DB URI, so it's safe to hide from ConfigMap.

k8s/02-backend-configured.yaml -> desktop
How to run

```
# 1) Ensure namespace exists (creates it if missing)
kubectl apply -f k8s/00-namespace.yaml
# -> creates namespace devops-part3 as defined in the YAML (no-op if already present)
```

Create ConfigMap & Secret

```
kubectl apply -f k8s/configmap.yaml
kubectl apply -f k8s/secret-mongo.yaml
```

Check:

```
kubectl -n devops-part3 get configmap app-config -o yaml
kubectl -n devops-part3 get secret mongo-secret -o yaml   # notice base64 values

# 4) Re-deploy backend using the Deployment that references the ConfigMap/Secret
kubectl apply -f k8s/02-backend-configured.yaml

# 4b) If pods do not automatically recycle to pick up new env vars, force a rolling restart:
kubectl -n devops-part3 rollout restart deployment/backend

# 5) Verify the envs inside a backend pod (target a pod, not a deployment)
# This picks the first backend pod name automatically and prints environment vars that contai
kubectl -n devops-part3 exec -it $(kubectl -n devops-part3 get pod -l app=backend -o jsonpath
# Expected output (example):
# MONGO_HOST=mongo
# MONGO_URI=mongodb://mongo:27017/taskdb
#

# 6) Check Services - useful to ensure frontend svc exists and ports are correct
kubectl -n devops-part3 get svc frontend -o wide
# -> shows ClusterIP, ports, and target selectors for the frontend service

# 7) Test the full flow: port-forward frontend (you can run this in a separate terminal)
kubectl -n devops-part3 port-forward svc/frontend 5173:5173
# -> opens local port 5173 and maps to the frontend service port 5173 inside cluster
# Now open in browser: http://localhost:5173
# Create a task in UI (frontend will POST to the configured API base)

# 8) Ensure backend is reachable from your host (port-forward backend in another terminal if
kubectl -n devops-part3 port-forward svc/backend 5000:5000
# -> now API endpoint is reachable on http://localhost:5000 (use curl or browser)
```

```
# 9) Confirm backend logs to see incoming requests and DB writes
kubectl -n devops-part3 logs -l app=backend --tail=200
# -> shows recent backend logs. If you see requests when creating tasks in UI, flow works.
```

Ingress

Chapter 1: Recap of Service Types (ClusterIP, NodePort, Port-Forward)

Before diving into Ingress, let's pause and remember how we have been exposing our applications so far.

In Kubernetes, Pods are short-lived and their IPs keep changing. That's why we created Services - they act as a stable entry point for a group of Pods.

But Services themselves come in different flavors, and each has its pros/cons. Let's go one by one.

ClusterIP - the default

What it is:

By default, when you create a Service, Kubernetes gives it a ClusterIP.

This IP is only accessible inside the cluster network.

```
kubectl get svc
```

NAME TYPE CLUSTER-IP PORT

backend ClusterIP 10.96.183.152 5000/TCP

Here, the backend has ClusterIP 10.96.183.152.

But try curl http://10.96.183.152:5000 from your laptop -> it won't work.

Use case:

Perfect for Pod-to-Pod communication (backend -> MongoDB).

Useless if you want to expose something to the outside world.

NodePort - the first step towards exposure

What it is:

NodePort exposes the Service on a port between 30000-32767 on each cluster node.

Example:

```
kubectl get svc
```

NAME TYPE CLUSTER-IP PORT(S)

backend NodePort 10.96.183.152 5000:31000/TCP

Now, you can hit http://<NodeIP>:31000/api/tasks from your browser.

Problem:

Each service = separate port.

If you have frontend, backend, and auth services, users would need to remember:

http://<IP>:31000

http://<IP>:32000

http://<IP>:32500

Totally unrealistic for real-world apps.

Engagement question:

"Would you enjoy browsing Amazon if you had to remember amazon.com:31000/cart and amazon.com:32000/payments?" (Students will laugh -> you land the point.)

Port-forward - the dev shortcut
What it is:
A temporary tunnel from your laptop into the cluster.
Example:

```
kubectl port-forward svc/backend 5000:5000
```

Now you can access backend at http://localhost:5000/api/tasks.
Great for debugging, but you can't expect 1,000 customers to each set up a secret wire.
Problem:
Works only while the command is running. Not shareable, not scalable, not for production.

The Problem Statement - Why NodePort Doesn't Scale
1. NodePort means "weird URLs" (2-3 mins)
Every Service exposed via NodePort gets a random high port between 30000-32767.
Example for our MERN app:
Backend -> http://192.168.99.101:31000/api/tasks
Frontend -> http://192.168.99.101:32000/
Imagine you ask a customer to open your app and they see:
"Hey, please go to http://192.168.99.101:32000/ for the UI, and if you need APIs, use http://192.168.99.101:31000/api."
Would any customer take that seriously? Absolutely not.
Analogy:
Think about restaurants. Normally, you go to pizza.com or burger.com. Easy.
But what if the owner said:
"Pizza is at pizza.com:31000"
"Burgers are at pizza.com:32000"
"Drinks are at pizza.com:32767"
You'd never return to that restaurant!

2. Too many services, too many ports (2-3 mins)
In a real-world microservices app, you don't have just a frontend and backend.
You might have:
Auth service
Payments service
Notifications service
File upload service
Analytics service
If each one needs a separate NodePort, you'd end up with:
:31000
:32000
:32500
:32650
...and so on.
This is like a house with 15 separate entrances - you'd constantly forget which door leads to which

room.

3. NodePort doesn't support domains (2-3 mins)
NodePort is IP + port based.
So if your cluster node has IP 192.168.99.101, you must access it like:
http://192.168.99.101:31000
But in the real world, customers expect:
http://myapp.com
or
https://shop.amazon.com/cart
With NodePort, there's no built-in support for domain names or path-based routing.
You'd need to put another reverse proxy (like Nginx) in front anyway - which defeats the purpose.
4. No SSL termination (1-2 mins)
Today, every serious website uses HTTPS.
But NodePort Services don't give you a straightforward way to add SSL certificates and terminate HTTPS.
You'd have to do it manually with a load balancer or external reverse proxy.
So NodePort is okay for labs and demos, but not production-grade.

Deep Dive - What is Ingress?
1. Simple definition (1-2 mins)
Ingress is like a smart router at the edge of your Kubernetes cluster.
It listens on standard web ports (80 for HTTP, 443 for HTTPS).
It takes incoming requests and, based on rules you define, sends them to the correct Service.
Instead of exposing every service separately, you just create one entry point.
2. Example with our MERN app (2-3 mins)
Without Ingress:
Backend -> http://192.168.99.101:31000/api/tasks
Frontend -> http://192.168.99.101:32000/
With Ingress:
One clean domain -> http://myapp.local
Ingress rules decide:
/ -> frontend service
/api -> backend service
So users don't need to care about IPs or ports. They just use:
http://myapp.local/
http://myapp.local/api/tasks

Real-world example: Amazon.com (2-3 mins)
Let's bring it home with something everyone knows:
When you go to amazon.com, you never type:
amazon.com:31000/cart
amazon.com:32000/payments
No. You just type:
amazon.com/ -> homepage service

amazon.com/cart -> cart service
amazon.com/payments -> payments service
amazon.com/profile -> user service
All under one domain.
That's exactly what Ingress enables in Kubernetes.
Without Ingress, Amazon would be unusable. Imagine if they gave you NodePorts:
amazon.com:31000/cart
amazon.com:32000/payments
You'd never shop there.
Key benefits of Ingress (1-2 mins)
List them out clearly (students can note them down):
[ok] One clean entry point -> no more multiple ports.
[ok] Path-based routing -> /api, /auth, /cart, etc.
[ok] Host-based routing -> shop.example.com vs blog.example.com.
[ok] SSL termination -> easy HTTPS setup.
[ok] Load balancing -> distribute requests across Pods.
[ok] Scalable -> add more services without exposing new ports.

The Flow of Traffic - Browser -> Ingress -> Services -> Pods
Start with a simple user story (1-2 mins)
Imagine a user named Ravi opens his browser and types:
http://myapp.local/api/tasks
From his perspective, it looks like magic - type a URL, get a webpage or JSON response.
But behind the scenes, a lot of steps happen. Let's break them down carefully.

Browser (http://myapp.local/api/tasks)

Ingress Controller (NGINX)

```
Rules:
  /     -> frontend Service (5173)
  /api  -> backend Service (5000)
```

Kubernetes Services
- frontend
- backend

Pods (containers)
React / Node.js / Mongo

3. Step-by-step walkthrough (4-5 mins)
Step 1 - Browser sends request
Ravi types http://myapp.local/api/tasks.

His browser makes an HTTP GET request with Host header = myapp.local.
Step 2 - Reaches Ingress Controller
The request enters the Kubernetes cluster through the Ingress Controller (like NGINX).
Think of this as the traffic police station at the city gate.
Step 3 - Ingress rules are checked
The Ingress Controller looks at the YAML-defined rules:
If path = / -> go to frontend service.
If path = /api -> go to backend service.
Since Ravi typed /api/tasks, the controller forwards to the backend service.
Step 4 - Service forwards to Pod
The backend service knows which Pods are running backend containers (via labels).
It load-balances the request to one of the Pods.
Step 5 - Pod handles the request
The Node.js app inside the Pod processes the /api/tasks request.
It may query MongoDB, prepare a JSON response, and send it back.
Step 6 - Response flows back
The Pod returns the response -> Service -> Ingress Controller -> Browser.
Ravi sees the result in his browser.
This full round trip happens in milliseconds.

Hands-On - Installing NGINX Ingress Controller (Minikube)
The Minikube magic - addons (1-2 mins)
Minikube makes life easier by providing some components as addons.
Instead of manually applying YAMLs, you just enable them.
For Ingress, Minikube has a built-in addon.
Command:

```
minikube addons enable ingress
```

This tells Minikube:
"Please install the NGINX Ingress Controller in my cluster."
Verifying installation (2-3 mins)
After enabling, you need to confirm it actually worked.
Run:

```
kubectl get pods -n ingress-nginx
```

Expected output:
NAME READY STATUS RESTARTS AGE
ingress-nginx-controller-5f56f6dfcd-2c9sv 1/1 Running 0 25s
Breakdown:
Namespace: ingress-nginx is where the controller lives.
Pod name: ingress-nginx-controller-xxxxx (random suffix).
READY: 1/1 -> means 1 container is running properly.
STATUS: Running.
If you see CrashLoopBackOff or Error, something went wrong (maybe low memory).

4. Inspect everything deployed (2-3 mins)
Run:

```
kubectl get all -n ingress-nginx
```

Typical output:
NAME READY STATUS RESTARTS AGE
pod/ingress-nginx-controller-5f56f6dfcd-2c9sv 1/1 Running 0 1m

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
service/ingress-nginx-controller NodePort 10.96.183.200 <none> 80:30523/TCP,443:32444/TCP 1m

NAME READY UP-TO-DATE AVAILABLE AGE
deployment.apps/ingress-nginx-controller 1/1 1 1 1m
Explain what you see:
Pod = the actual NGINX process running.
Service = exposes the controller itself.
Deployment = ensures controller stays alive (self-heals if Pod crashes).

Writing the Ingress YAML (Path-Based Routing)

Set the stage (1-2 mins)
"We've got the Ingress Controller running. But remember - without rules, it's just standing at the gate with no signboards.
Now, let's create those rules so it knows where to send / traffic and where to send /api traffic."
We'll write a YAML file named:
k8s/04-ingress.yaml

yaml file explanation ->
Line-by-line breakdown (7-8 mins)

```
apiVersion: networking.k8s.io/v1
```

Ingress belongs to the networking API group in Kubernetes.
v1 is stable and supported across all modern clusters.

```
kind: Ingress
```

We're declaring this object as an Ingress resource.

```
metadata: name: mern-ingress
```

The Ingress resource's name inside the cluster.
Convention: use the app name + -ingress.

namespace: devops-part3
Ingress should live in the same namespace as our app.
annotations (optional tweaks):
nginx.ingress.kubernetes.io/rewrite-target: /
Ensures correct path forwarding (strip and rewrite if needed).
nginx.ingress.kubernetes.io/proxy-body-size: "10m"
Increases upload size limit (e.g., if frontend uploads files).
spec.ingressClassName: nginx
This tells Kubernetes: "Use the NGINX Ingress Controller for this Ingress."
If you had multiple controllers, this chooses the right one.
rules -> host: merndemo.local
This is the domain name users will type.
On our laptop, we'll map merndemo.local to the Minikube IP via /etc/hosts.
http.paths
This is where the routing magic happens.
path: /api -> send requests to backend service on port 5000.
path: / -> send all other requests to frontend service on port 5173.
Explain that pathType: Prefix means /api matches /api, /api/tasks, /api/anything....

Connecting to our MERN app (3-4 mins)
Our frontend runs on port 5173.
Our backend runs on port 5000.
Without Ingress, we needed separate NodePorts.
With Ingress, both are now available from one single domain.
So:
http://merndemo.local/ -> React app (frontend).
http://merndemo.local/api/tasks -> backend API (Node.js).

Important adjustment - frontend build (3-4 mins)
Very important step.
Our frontend code (React/Vite) needs to call the backend correctly through Ingress.
In local dev, we used VITE_API_URL=http://backend:5000.
But with Ingress, the backend is available at /api.
So we must rebuild the frontend with:

```
docker build -t frontend:local  --build-arg VITE_API_URL=/api ./client
minikube image load frontend:local
kubectl -n devops-part3 rollout restart deploy/frontend
```

Why?
Because then when frontend code makes fetch("/api/tasks"), the request goes to
http://merndemo.local/api/tasks.
Ingress routes it to backend automatically.
[ok] No CORS issues.

Apply the Ingress resource
Now apply the YAML:

```
kubectl apply -f k8s/04-ingress.yaml
```

Check status:

```
kubectl -n devops-part3 get ingress
```

Expected output:
NAME CLASS HOSTS ADDRESS PORTS AGE
mern-ingress nginx merndemo.local 192.168.49.2 80 10s
HOSTS -> merndemo.local (our fake domain).
ADDRESS -> Minikube IP.
PORTS -> 80 (HTTP).

Map host to Minikube IP (2-3 mins)
Get the Minikube IP:

```
minikube ip
```

Suppose it returns 192.168.49.2.
Now edit /etc/hosts (on your laptop) as root:
192.168.49.2 merndemo.local
Now your browser will know that merndemo.local should go to the Minikube cluster.

Test in the browser (2-3 mins)
Open browser and test:
http://merndemo.local/ -> should load frontend UI.
http://merndemo.local/api/tasks -> should hit backend API.
Also test with curl:

```
curl http://merndemo.local/
curl http://merndemo.local/api/tasks
```

final flow
Browser (http://merndemo.local/api/tasks)

Ingress Controller (NGINX)
Rule: /api -> backend service

Backend Service (5000)

Backend Pod (Node.js)

MongoDB Pod

All commands to run ->

```
# =========================
# 1) Build images locally
# =========================

# Backend (server.js must use 0.0.0.0)
docker build -t backend-new:local -f server/Dockerfile server

# Frontend (static build with serve -s dist -l 5173)
docker build -t frontend-new:local -f client/Dockerfile --build-arg VITE_API_URL=/api client

# =========================
# 2) Load images into Minikube
# =========================
minikube image load backend-new:local
minikube image load frontend-new:local

# =========================
# 3) Apply Kubernetes YAMLs
# (namespace, config, secrets, deployments, services, ingress)
# =========================
kubectl apply -f k8s/00-namespace.yaml
kubectl apply -f k8s/app-config.yaml
kubectl apply -f k8s/mongo-secret.yaml
kubectl apply -f k8s/01-mongo.yaml
kubectl apply -f k8s/02-backend.yaml
kubectl apply -f k8s/03-frontend.yaml
kubectl apply -f k8s/04-ingress.yaml

# =========================
# 4) Point deployments to new images & restart
# =========================
kubectl -n devops-part3 set image deploy/backend  backend=backend-new:local
kubectl -n devops-part3 set image deploy/frontend frontend=frontend-new:local

kubectl -n devops-part3 rollout status deploy/backend
kubectl -n devops-part3 rollout status deploy/frontend

# =========================
# 5) Open tunnel for Ingress (keep this terminal OPEN)
# =========================
minikube tunnel
```

Minikube creates a network bridge between your Windows machine (localhost) and the Ingress controller running inside the Kubernetes cluster.
Without it, traffic from http://merndemo.local never reaches the NGINX Ingress pod.
With it, Minikube listens on 127.0.0.1:80 (and 443 if you use HTTPS) and forwards that traffic straight into the cluster.
That's why:
You keep it open in one terminal (it's actively forwarding).
Your browser on Windows can reach merndemo.local -> Ingress -> frontend/backend.
Think of it as a tunnel pipe that connects your PC's port 80 with the cluster's port 80.
Without the tunnel -> browser requests never find the app.
With the tunnel -> you can type http://merndemo.local/ and it "just works."

```
# =========================
# 6) Add hosts entry (edit with admin rights)
# File: C:\Windows\System32\drivers\etc\hosts
# =========================
```

```
127.0.0.1  merndemo.local

# =========================
# 7) Test in PowerShell
# =========================
curl.exe -v http://merndemo.local/
curl.exe -v http://merndemo.local/api/tasks

# Then open in browser:
# http://merndemo.local/
```

Storage & Persistence in Kubernetes
Concepts in Simple Words
The Problem with emptyDir
Right now, your Mongo Deployment uses:

```
volumes:
```

- name: mongo-data

```
    emptyDir: {}
```

emptyDir is like a temporary notebook a student keeps in class.
As long as the student (Pod) is in class, the notebook is there.
But the moment the student leaves (Pod restarts, node crash), the notebook is thrown away.
This means:
If Mongo pod is deleted or rescheduled -> all data is lost.
Bad for production DBs where persistence is critical.

The Idea of Persistent Storage
Kubernetes separates compute (Pods) from storage.
Pods are ephemeral: they can die, restart, or move to another node anytime.
Storage should be persistent: data must survive pod restarts.
To solve this:
PersistentVolume (PV)
A chunk of actual storage in the cluster (disk, cloud volume, NFS, etc.).
Think of it like a locker in a school.
Independent of Pods.
PersistentVolumeClaim (PVC)
A request by a Pod for storage.
Like a student (Pod) saying: "I need a locker with at least 1GB space."
Kubernetes matches the request (PVC) to available lockers (PV).
StorageClass (optional, advanced)
Defines how storage is provisioned (SSD vs HDD, cloud disk vs local).
For now in Minikube, default storage class auto-creates PVs for us.
Why This Matters in Real Apps
In real production apps, databases cannot lose data.
PVC ensures:
Data is not tied to Pod lifecycle.
If Mongo pod is deleted, the new pod reuses the same PVC -> same data.

Without this, your MERN app would "forget" all tasks after every restart -> useless.

Flows / Step-by-Step Setup
Here's the step-by-step flow:
Check current Mongo Deployment
Notice it uses emptyDir -> ephemeral.
Demo: create a task, delete Mongo pod -> data disappears.

Create a PVC

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
```

name: mongo-pvc
namespace: devops-part3

```
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Update Mongo Deployment
Replace emptyDir with:

```
volumes:
```

- name: mongo-data

```
    persistentVolumeClaim:
      claimName: mongo-pvc
```

Apply changes

```
kubectl apply -f k8s/01-storage.yaml
kubectl -n devops-part3 rollout restart deploy/mongo
```

Verify
Run kubectl -n devops-part3 get pvc -> should show Bound.
Run kubectl -n devops-part3 describe pvc mongo-pvc -> check volume binding.

Demo Flow
Step 1: Show with emptyDir
Open your frontend at http://merndemo.local.
Create a new task in the UI (e.g., "Persistence Test 1").
Refresh the page -> show the task is there.
Now delete the Mongo pod:

```
kubectl -n devops-part3 delete pod -l app=mongo
kubectl -n devops-part3 get pods -w
```

Refresh the UI -> task is gone.
[ok] You've shown data loss with emptyDir.

Step 2: Add PVC
YAMLs to add/change:
PVC file (k8s/02-mongo-pvc.yaml)

Mongo Deployment -> replace emptyDir with PVC:

Apply new storage config

```
kubectl apply -f k8s/mongo-pvc.yaml
kubectl apply -f k8s/03-mongo.yaml
kubectl -n devops-part3 rollout restart deploy/mongo
kubectl -n devops-part3 rollout status deploy/mongo
```

Check PVC:

```
kubectl -n devops-part3 get pvc
```

-> Should say Bound

Step 4: Show persistence
Go to your UI (http://merndemo.local).
Create a new task (e.g., "Persistence Test 2").
Refresh -> task exists.
Delete Mongo pod again:

```
kubectl -n devops-part3 delete pod -l app=mongo
```

Refresh UI -> task is STILL there.
[ok] You've shown persistence with PVC.