**Step-by-Step: Accessing an Existing OpenShift Cluster   
  
Install the oc CLI on Your Local Machine**

**STEP 1: On macOS/Linux:**

If you don't already have the OpenShift CLI installed:   
curl -LO <https://mirror.openshift.com/pub/openshift-v4/clients/ocp/latest/openshift-client-linux.tar.gz>   
  
tar -xvzf openshift-client-linux.tar.gz   
  
sudo mv oc kubectl /usr/local/bin/   
Replace linux with mac if you're on macOS.

**STEP 2: Get the API Server URL and Web Console URL**  
Ask the cluster administrator to share:   
API Server URL (used with oc login):   
https://api.<cluster-domain>:6443   
Web Console URL (for browser access):   
[https://console-openshift-console.apps](https://console-openshift-console.apps/).<cluster-domain>   
  
**STEP 3: Get Your Login Credentials**

You’ll need either of the following:   
A username and password (e.g., kubeadmin or SSO/LDAP account)   
A token (a long alphanumeric string)   
A kubeconfig file   
  
**STEP 4: Log In Using oc CLI**   
  
Option A: With username and password   
oc login https://api.<cluster-domain>:6443 -u <username> -p <password>   
  
Option B: With token   
oc login https://api.<cluster-domain>:6443 --token=<token>   
After this, you'll see:   
Login successful.   
You have access to 56 projects, the list has been suppressed. You can list all projects with 'oc projects'   
  
**STEP 5: Verify Your Connection**

Run:   
oc whoami   
You should see your username.   
  
Then try:   
oc get nodes   
To check if your CLI can reach the cluster.   
  
**STEP 6: Access the Web Console**

Open your browser.   
Visit:   
[https://console-openshift-console.apps](https://console-openshift-console.apps/).<cluster-domain>   
Log in with the same credentials you used for the CLI.   
  
Now you can:   
Use the CLI for automation and scripting (oc get, oc apply, oc logs, etc.)   
Use the Web Console for visual workflows (creating apps, managing workloads, pipelines, monitoring, etc.)   
  
  
**Complete OpenShift traffic flow — from Route to Service to Pod**   
  
**SCENARIO**

We are deploying two replicas of a basic Nginx app:   
Route: Exposes the app to external traffic.   
Service: Load balances traffic across the pods.   
Deployment: Runs 2 identical pods.   
  
Load Balancing: Handled at the Service and Route level.   
  
**FLOW OVERVIEW**   
    
Client (browser)   
↓   
 OpenShift Router (HAProxy via Route)   
↓   
 Service (ClusterIP)   
↓   
 Pod 1 (Nginx)   
Pod 2 (Nginx)   
  
**YAMLs for All Components**   
  
**1. Deployment (2 Pods)**   
apiVersion: apps/v1   
kind: Deployment   
metadata:   
name: nginx-deployment   
labels:   
app: nginx   
spec:   
replicas: 2   
selector:   
matchLabels:   
app: nginx   
template:   
metadata:   
labels:   
app: nginx   
spec:   
containers:   
- name: nginx   
image: nginx:1.20   
ports:   
- containerPort: 80   
  
**2. Service (ClusterIP)**   
apiVersion: v1   
kind: Service   
metadata:   
name: nginx-service   
spec:   
selector:   
app: nginx # Matches pods from deployment   
ports:   
- protocol: TCP   
port: 80 # Service port   
targetPort: 80 # Pod port   
type: ClusterIP # Default   
  
**3.Route (External Access)**   
apiVersion: [route.openshift.io/v1](https://web.telegram.org/a/route.openshift.io/v1)   
kind: Route   
metadata:   
name: nginx-route   
spec:   
host: [nginx-example.apps.example.com](https://web.telegram.org/a/nginx-example.apps.example.com)  # Change to your domain   
to:   
kind: Service   
name: nginx-service   
port:   
targetPort: 80   
tls:   
termination: edge   
  
This Route exposes the app externally on a domain like:   
[http://nginx-example.apps.example.com](http://nginx-example.apps.example.com/)   
  
**TRAFFIC FLOW (STEP-BY-STEP)**   
  
User enters [http://nginx-example.apps.example.com](http://nginx-example.apps.example.com/) in the browser.   
OpenShift’s Router (HAProxy) looks for a matching Route.   
It finds nginx-route, which points to the Service nginx-service.   
Service forwards the traffic to any of the two pods labeled app: nginx.   
Traffic is round-robined by the Service.   
Pod responds, and response flows back through the same path.   
  
What is the OpenShift Router?   
It is OpenShifts Ingress Controller, based on HAProxy.   
Deployed automatically in every OpenShift cluster.   
Runs as a Pod inside the openshift-ingress namespace.   
Handles incoming traffic from outside and routes it to the correct Service inside the cluster.

**Step-by-Step: How OpenShift Router Works**

When the OpenShift cluster is installed:   
The installer deploys the OpenShift Router.   
It sets up:   
A Deployment with HAProxy router pods   
A Service (to expose those pods internally)   
A LoadBalancer or NodePort (to expose the router externally)   
  
To see the router:   
oc get pods -n openshift-ingress   
  
When OpenShift is installed, it is configured with a default wildcard domain like:   
\*.[apps.example.com](https://web.telegram.org/a/apps.example.com)   
  
This means:   
You can choose any subdomain like:   
[nginx-example.apps.example.com](https://web.telegram.org/a/nginx-example.apps.example.com)   
[myapp.apps.example.com](https://web.telegram.org/a/myapp.apps.example.com)   
[frontend.apps.example.com](https://web.telegram.org/a/frontend.apps.example.com)   
These all point to the same external IP of the OpenShift Router.   
  
You can check this wildcard domain by running:   
oc get [ingresses.config.openshift.io](https://web.telegram.org/a/ingresses.config.openshift.io) cluster -o jsonpath='{.spec.domain}'   
  
**Application Context**   
You usually name the route after your app or service.   
Examples:   
App Name Suggested Route Host   
nginx [nginx.apps.example.com](https://web.telegram.org/a/nginx.apps.example.com)   
frontend [frontend.apps.example.com](https://web.telegram.org/a/frontend.apps.example.com)   
orders-api [orders-api.apps.example.com](https://web.telegram.org/a/orders-api.apps.example.com)   
You define this manually to make it meaningful and unique.   
DNS and Routing:   
Your public DNS must have a wildcard record like this:   
\*.[apps.example.com](https://web.telegram.org/a/apps.example.com) → [OpenShift Router Load Balancer IP]   
If that's configured, any route you create under that domain becomes immediately accessible.   
  
  
**External Access via DNS + Load Balancer**   
Your DNS must map:   
\*.[apps.example.com](https://web.telegram.org/a/apps.example.com) → LoadBalancer IP of the OpenShift Router   
This LoadBalancer is either:   
Provisioned automatically   
Manually set up   
  
To find it:   
oc get svc -n openshift-ingress   
Look for:   
NAME TYPE CLUSTER-IP EXTERNAL-IP   
router-default LoadBalancer 172.x.x.x 203.x.x.x   
  
That 203.x.x.x is what your \*.[apps.example.com](https://web.telegram.org/a/apps.example.com) DNS wildcard should point to.

**Real-World Analogy**   
  
Imagine:   
Your OpenShift cluster is a gated colony (private).   
The Router is the security gate.   
The LoadBalancer is the colony’s main entry road with a unique address.   
  
So:   
A visitor (browser) wants to visit [nginx.apps.example.com](https://web.telegram.org/a/nginx.apps.example.com)   
DNS says: this domain maps to [203.0.113.10](https://web.telegram.org/a/203.0.113.10) (LoadBalancer IP)   
The LoadBalancer routes the request to the OpenShift Router   
The Router checks the Route rules and sends it to the correct Service   
The Service picks a healthy Pod and sends the traffic   
  
**Flow:**   
Browser request → [nginx.apps.example.com](https://web.telegram.org/a/nginx.apps.example.com)   
↓   
DNS resolves → [203.0.113.10](https://web.telegram.org/a/203.0.113.10)   
↓   
Hits LoadBalancer   
↓   
Goes to OpenShift Router (HAProxy)   
↓   
Looks at Route → finds nginx-service   
↓   
nginx-service → forwards to a Pod with label app=nginx   
↓   
Pod responds   
  
**Who Provides the LoadBalancer?**   
In Cloud (like AWS, Azure, GCP):   
OpenShift uses cloud APIs to automatically create a real LoadBalancer   
It gets a public IP and routes traffic internally   
In On-Premise (bare metal):   
No automatic LoadBalancer available   
You manually expose the router using:   
MetalLB   
Or change Service type to NodePort and use external LB (like F5, Citrix, etc.)   
  
  
  
**Scenario: Blue/Green or Canary Deployment**   
  
You have:   
v1: current stable version of the app   
v2: new version under test   
You want:   
80% of traffic to go to v1   
20% to go to v2   
  
**1.Deployments for Two Versions**   
  
**v1 Deployment (Blue)**   
  
apiVersion: apps/v1   
kind: Deployment   
metadata:   
name: myapp-v1   
labels:   
version: v1   
spec:   
replicas: 2   
selector:   
matchLabels:   
app: myapp   
version: v1   
template:   
metadata:   
labels:   
app: myapp   
version: v1   
spec:   
containers:   
- name: myapp   
image: nginx:1.20   
ports:   
- containerPort: 80   
  
**v2 Deployment (Green / Canary)**   
  
apiVersion: apps/v1   
kind: Deployment   
metadata:   
name: myapp-v2   
labels:   
version: v2   
spec:   
replicas: 1   
selector:   
matchLabels:   
app: myapp   
version: v2   
template:   
metadata:   
labels:   
app: myapp   
version: v2   
spec:   
containers:   
- name: myapp   
image: nginx:1.21   
ports:   
- containerPort: 80   
  
  
**Two Separate Services**   
  
**Service for v1**   
  
apiVersion: v1   
kind: Service   
metadata:   
name: myapp-v1

spec:   
selector:   
app: myapp   
version: v1   
ports:   
- port: 80   
targetPort: 80   
  
  
**Service for v2**   
  
apiVersion: v1   
kind: Service   
metadata:   
name: myapp-v2   
spec:   
selector:   
app: myapp   
version: v2   
ports:   
- port: 80   
targetPort: 80   
  
**3. Route with Weighted Traffic Split**   
  
apiVersion: [route.openshift.io/v1](https://web.telegram.org/a/route.openshift.io/v1)   
kind: Route   
metadata:   
name: myapp-route   
spec:   
host: [myapp-bluegreen.apps.example.com](https://web.telegram.org/a/myapp-bluegreen.apps.example.com)   
to:   
kind: Service   
name: myapp-v1   
weight: 80   
alternateBackends:   
- kind: Service   
name: myapp-v2   
weight: 20   
port:   
targetPort: 80   
tls:   
termination: edge   
  
  
**What Happens When a User Accesses the App**   
  
User visits [http://myapp-bluegreen.apps.example.com](http://myapp-bluegreen.apps.example.com/)   
OpenShift’s HAProxy Router receives the request   
HAProxy checks the route:   
80% chance → forwards to myapp-v1 service → to one of the v1 pods   
20% chance → forwards to myapp-v2 service → to the v2 pod