BY FENIL GAJJAR

KUBERNETES DAILY TASKS

KUBERNETES
INGRESS TO GATEWAY API

- COMPEREHENSIVE GUIDE
- THEOYR + PRACTICAL TASKS
- REAL TIME SCENARIO TASKS
- END TO END DOC

Contact Us:

fenilgajjar.devops@gmail.com

github.com/Fenil-Gajjar

🔀 Kubernetes Ingress ➡️

Gateway API Migration

Real-World Implementation

& Best Practices 🚀

👋 Welcome to the Ingress ➡️ Gateway API Migration Journey!

Hello Engineers!
First of all — **a massive thank you** to all of you who've followed my technical documentation series so far 🙏. Whether you've explored my breakdowns on **Kubernetes**, **DevOps tools**, or most recently, the **deep-dive into the Gateway API** — your support, engagement, and feedback have been incredible. It truly fuels my commitment to deliver more hands-on, production-level insights for real-world engineering problems.

Now, let's talk about this doc...

🚀 What's Inside This Doc?

In this documentation, we're shifting gears into a highly practical and **impactful DevOps use case**:

> **Migrating from legacy Kubernetes Ingress to the powerful Gateway API** — all while using the **NGINX Gateway Controller** on a **Kops-managed Kubernetes cluster**.

Here's what you'll uncover:

- ✅ A clear understanding of **why migrating to Gateway API is essential**
- ✅ **Key precautions** before and during the migration process

- ✅ Multiple **real-time scenario-based tasks** showing actual YAMLs, configs, and routing flows
- ✅ In-depth **implementation using NGINX**
- ✅ **Troubleshooting techniques** you'll need in production
- ✅ And finally — bulletproof **best practices** for operating Gateway API the right way in a real DevOps environment

Whether you're a **DevOps engineer, platform architect, or SRE**, this doc will guide you with clarity, context, and confidence to take your networking layer to the next level.

📚 Built for the Real World — Not Just Theory

This isn't just another high-level comparison. It's a hands-on, YAML-heavy, scenario-backed deep dive, written from experience, tested on real workloads, and structured to **bridge the gap between theory and production reality**.

And the best part? It's built entirely with:

- 🔹 Open-source tooling
- 🔹 Real infra (via **Kops**)
- 🔹 Practical, scalable, secure architecture decisions

📌 Stay Tuned for More!

This is just one part of a much bigger series where I'll continue exploring:

So if this excites you — **stick around, share your feedback, and keep learning with me.**

Let's make Kubernetes more powerful, one layer at a time.

Migrating from Ingress to Gateway API: What Happens and Why It Matters

📌 Background

Ingress has long been the standard for managing external access to services in Kubernetes. It's simple, widely adopted, and works well for basic use cases. However, as Kubernetes workloads and architectures grow more complex, **Ingress falls short** in areas like:

- **Advanced traffic routing**
- **Extensibility**
- **Separation of concerns**
- **Cross-team collaboration**

This is where **Gateway API** steps in — a Kubernetes-native, flexible, and extensible standard designed to **replace and enhance** Ingress for modern networking needs.

🚨 So What Happens When Most of Your Workloads Are Still in Ingress?

1. Legacy Lock-in

You're tied to Ingress-specific annotations and behaviors, often vendor-dependent (e.g., NGINX, Traefik). This can make your workloads:

- Harder to port
- Less maintainable
- Less cloud-native

2. Limited Traffic Control

With Ingress, your routing capabilities are basic (host/path-based). If your architecture needs:

- Traffic splitting
- Header-based routing
- Weighted canary deployments

...you'll quickly hit a wall.

3. Poor Multi-Tenancy Support

Ingress doesn't support separation of roles well — you often end up mixing developer, platform, and security concerns in one big Ingress YAML. Gateway API introduces **GatewayClasses**, **Gateways**, and **Routes**, offering:

- Role separation
- Cleaner delegation
- Better RBAC control

4. Annotation Hell

Ingress relies heavily on annotations for anything advanced. Different Ingress controllers interpret these differently. This can lead to:

- Configuration drift
- Unexpected behavior
- Difficult troubleshooting

🔁 When You Migrate to Gateway API, You Get:

- ✅ **Standardization** across controllers
- ✅ **Modern APIs** built with extensibility in mind
- ✅ **Better security posture** via route delegation
- ✅ **More observability and debugging tools**
- ✅ **Better compatibility** with service mesh patterns (e.g., Istio, Linkerd)

🔄 Realistic Migration Outcomes

When migrating existing Ingress-based workloads to Gateway API, **you're not just rewriting YAMLs**. You're:

- Re-architecting your traffic flow
- Redefining who owns what in your platform (Dev vs Ops)
- Potentially **changing your ingress controller** to one that supports Gateway API (e.g., `istio`, `GKE Gateway Controller`, `Kong`, `Contour`, `NGINX Gateway`)

Your workloads:

- Will need **new Gateway API resource definitions** (GatewayClass, Gateway, HTTPRoute, etc.)

- May need **testing and gradual rollout** to avoid downtime

- Should be validated for **parity in behavior** (what used to work in Ingress should behave the same in Gateway)

Why Migrate from Ingress to Gateway API?

As Kubernetes evolves and becomes the foundation for production workloads across teams and organizations, **Ingress is no longer enough**. While it served well for simple HTTP routing, today's infrastructure demands **more power, flexibility, and maintainability** — and that's where Gateway API comes in.

Let's break down **why you should consider migrating**:

🔧 1. Ingress Is Too Simple for Modern Use Cases

- Ingress only supports **basic routing** like host- and path-based routing.
- Advanced use cases (like header-based routing, traffic splitting, canary deployments) require **controller-specific hacks or annotations**.
- Gateway API natively supports these advanced routing rules, with **first-class APIs**.

> ✅ **Gateway API provides rich routing features natively.**

🏗️ 2. Poor Extensibility in Ingress

- Ingress doesn't scale well when you need to **extend behavior or introduce new traffic control mechanisms**.

- You're stuck using **non-portable annotations**, which are controller-specific and error-prone.

✅ **Gateway API is built from the ground up to be extensible**, supporting CRDs for custom filters, authentication, etc.

👥 3. No Separation of Concerns in Ingress

- Ingress is a single resource controlled by both developers and platform teams.
- This creates conflicts, misconfigurations, and fragile pipelines.

✅ **Gateway API introduces a clean separation:**

- **GatewayClass** – Cluster-level admin config (infra team)
- **Gateway** – Gateway definition (platform team)
- **HTTPRoute** – Routing rules (application team)

This supports **multi-tenancy and delegation**, enabling **clear ownership boundaries**.

🔒 4. Better Security and RBAC Support

- In Ingress, anyone with access can modify any route.
- Gateway API allows **fine-grained permissions** — platform teams manage the Gateway, devs manage their own routes.

✅ **More secure and auditable configurations**, aligned with enterprise policies.

🔍 5. Standardized Observability & Status Reporting

- Ingress gives limited status reporting, and it's often controller-specific.
- Gateway API introduces a **standardized and expressive status model**, showing:
    - Route health
    - Attachment status
    - Gateway readiness

✅ Easier troubleshooting and monitoring.

📦 6. Vendor-Neutral & Controller-Agnostic

- Ingress behavior can vary wildly between controllers (NGINX, Traefik, ALB, etc.).
- Gateway API aims to be **vendor-neutral**, with well-defined conformance tests and features across all compliant controllers.

✅ Migrate once, avoid vendor lock-in.

🌱 7. Future-Proofing Your Cluster

- The Kubernetes community is **actively developing Gateway API**.
- Ingress is **not going away**, but it's not evolving either.
- Gateway API is where new innovation is happening (e.g., Mesh integration, gRPC support, richer APIs).

✅ Adopting Gateway API aligns your infrastructure with **Kubernetes networking's future**.

Precautions and Considerations Before Migrating from Ingress to Gateway API

Migrating from Ingress to Gateway API is not a simple “search and replace” task. It’s a **strategic shift** in how your applications handle traffic. To ensure a smooth and safe transition, here are the **key precautions and things to keep in mind** before and during the migration process.

🚧 1. Not All Ingress Controllers Support Gateway API (Yet)

Before starting the migration, **verify that your ingress controller supports Gateway API**. Popular controllers like:

- **NGINX** (via `nginx-gateway-fabric`)
- **Istio**
- **Contour**
- **Kong**
- **GKE Gateway Controller**

...are adding support, but it might not be feature-complete or stable in your environment.

✅ **Action:** Check compatibility of your current controller or plan to switch to one that supports Gateway API in production.

🔄 2. Gateway API Is Not a 1:1 Replacement

- Ingress and Gateway API **have different resource models**.
- A single Ingress may map to **multiple Gateway API resources** like `Gateway`, `HTTPRoute`, etc.
- Certain Ingress features (e.g., TLS termination, rewrite rules) must be **explicitly re-implemented** using Gateway filters or route-level settings.

✅ **Action:** Design your new resource layout carefully before migration.

🧪 3. Feature Parity Isn't Always Guaranteed

Some advanced features or annotations from Ingress:

- May **not be supported yet** in Gateway API
- Or may behave **differently across controllers**

Example:

- Path rewrites in NGINX via annotations might require a **custom Gateway filter** in Gateway API.

✅ **Action:** Test each route's behavior before and after migration to ensure parity.

🧭 4. Gradual Migration is Key — Don't "Lift and Shift"

- Avoid migrating everything at once.
- Instead, follow a **phased rollout**, such as:
    - Start with **a single service or namespace**
    - Use **side-by-side deployment** (Ingress + Gateway)
    - Validate with canary traffic or staging environments

✅ **Action:** Plan for a progressive rollout strategy.

🔐 5. TLS, Auth, and Cert Management Needs Rethinking

- TLS settings in Gateway API are **defined at the Gateway level**, not on routes like in Ingress.
- Cert-manager integrations and ACME configurations may need changes.

> ✅ **Action:** Revisit your TLS termination setup and confirm certs are correctly handled in Gateway API.

👥 6. Define Clear Ownership for New Resources

Gateway API introduces separation of concerns:

- `GatewayClass` – Cluster/network team
- `Gateway` – Platform team
- `HTTPRoute` – Application/dev team

If roles aren't clearly defined, **confusion and access issues** will occur.

> ✅ **Action:** Define and communicate ownership responsibilities across teams.

🔍 7. Observability and Monitoring Need Updates

- If you use dashboards, alerts, or log parsing for Ingress behavior, these will need to adapt to:
    - `Gateway` and `Route` status fields
    - New logs and metrics paths

> ✅ **Action:** Update observability stack to reflect Gateway API semantics.

🛡️ 8. Update RBAC Rules

Your existing RBAC setup may allow access to `Ingress` objects, but not:

- `GatewayClass`
- `Gateway`
- `HTTPRoute`

> ✅ **Action:** Review and update RBAC permissions to align with the new API structure.

🧰 9. Tooling and CI/CD Integration

- Your CI/CD pipelines may include Ingress YAMLs, testing logic, or validation scripts.

- These will all need **updates to support Gateway API resources**.

✅ **Action:** Prepare your toolchain and pipelines for the new structure.

🔄 Step-by-Step: Migrating from Ingress to Gateway API (NGINX Controller on Kops)

This section provides a **real-world migration scenario**, starting from a basic Ingress resource and moving to a Gateway API setup, with NGINX Gateway Controller handling the traffic.

> ✅ **Target Audience:** DevOps engineers managing their own Kops-based Kubernetes clusters.

🧱 1. Prerequisites

Before migration, ensure:

- ✅ You have a running Kubernetes cluster via **Kops**
- ✅ `kubectl` is configured and working
- ✅ You're using **NGINX Gateway Controller** (not classic NGINX Ingress controller)
- ✅ Gateway API CRDs are installed (we'll cover that below)

📦 2. Install the NGINX Gateway Controller (Gateway API version)

Here's how you install the NGINX Gateway API controller in your cluster:

```
kubectl apply -k
"github.com/nginxinc/nginx-gateway-kubernetes/deploy/kuberne
tes/overlays/gateway-api?ref=v1.1.0"
```

🔍 This installs:

- Gateway API CRDs
- NGINX Gateway Controller
- Necessary RBAC and deployments

📘 3. Original Ingress YAML Example

Here's a simple Ingress definition that exposes a backend service:

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

```
  name: my-app-ingress
  namespace: demo
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: myapp.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: my-app
            port:
              number: 80
```

📤 4. Migrate to Gateway API Resources

You'll now define the **GatewayClass**, **Gateway**, and **HTTPRoute** to replace the Ingress.

✅ a. Create a `GatewayClass` (NGINX-specific)

```
apiVersion: gateway.networking.k8s.io/v1
kind: GatewayClass
metadata:
  name: nginx
spec:
  controllerName: nginx.org/gateway-controller
```

Apply it:

```
kubectl apply -f gatewayclass.yaml
```

✅ b. Create a `Gateway` Resource

This defines the load balancer entry point.

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
```

```
metadata:
  name: nginx-gateway
  namespace: demo
spec:
  gatewayClassName: nginx
  listeners:
  - name: http
    protocol: HTTP
    port: 80
    hostname: "myapp.example.com"
    allowedRoutes:
      namespaces:
        from: Same
```

Apply it:

```
kubectl apply -f gateway.yaml
```

✅ c. Create an `HTTPRoute`

This is equivalent to the Ingress rule — it defines how requests are routed.

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: my-app-route
  namespace: demo
spec:
  parentRefs:
  - name: nginx-gateway
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /
    backendRefs:
    - name: my-app
      port: 80
```

Apply it:

```
kubectl apply -f httproute.yaml
```

🧪 5. Test the Migration

After applying everything:

- Confirm Gateway and HTTPRoute status:

```
kubectl get gateway -n demo

kubectl get httproute -n demo
```

- Test access via DNS or by port-forwarding temporarily:

```
kubectl port-forward svc/nginx-gateway 8080:80 -n
nginx-gateway

curl -H "Host: myapp.example.com" http://localhost:8080/
```

🔍 6. Feature Matching: Validate Parity

You must verify that the new Gateway API route **behaves exactly as the old Ingress**:

- Are paths routed correctly?
- Are headers preserved?
- Is TLS termination working (if configured)?
- Do redirects or rewrites work as expected?

> ⚠️ Some features in Ingress may require custom filters in Gateway API. NGINX Gateway Controller is still maturing, so test critical paths carefully.

📦 7. Decommission Old Ingress

Once you're confident everything works, clean up the old Ingress:

```
kubectl delete ingress my-app-ingress -n demo
```

🧠 **Bonus: TLS Termination (Optional)**

Want to terminate TLS at the Gateway level?

Update your Gateway like this:

```
listeners:
- name: https
  protocol: HTTPS
  port: 443
  hostname: "myapp.example.com"
  tls:
    certificateRefs:
    - kind: Secret
      name: myapp-tls-secret
  allowedRoutes:
    namespaces:
      from: Same
```

📌 TLS secrets must exist in the **same namespace as the Gateway**.

🧰 What You Need to Have Before Migrating to Gateway API

Implementing this migration requires preparation at **cluster**, **tooling**, and **team** levels. Below is a detailed checklist to guide your readiness:

🧠 Technical Understanding of Gateway API Concepts

Before you write any YAMLs, your team should be familiar with key Gateway API components:

- `GatewayClass` – Defines controller behavior (like a driver)
- `Gateway` – Represents the actual load balancer
- `HTTPRoute` – Defines routing rules for applications
- `ParentRefs`, `BackendRefs`, and `Listeners` – How traffic flows and is routed

> ✅ **Action:** Educate your team on Gateway API basics (docs, quickstart labs)

☁️ A Kops-Based Kubernetes Cluster Up and Running

You need a functioning cluster provisioned with Kops:

- Proper networking set up (VPC, subnets, DNS, etc.)
- Working `kubectl` configuration
- External DNS support if you're using domain names (e.g., `myapp.example.com`)

> ✅ **Action:** Confirm your cluster is reachable and can expose public endpoints.

🚪 NGINX Gateway Controller Installed

Make sure you're using the **Gateway API-compatible version**, not the classic `nginx-ingress` controller.

> ✅ Use this to install:

```
kubectl apply -k
"github.com/nginxinc/nginx-gateway-kubernetes/deploy/kuberne
tes/overlays/gateway-api?ref=v1.1.0"
```

✅ **Check:** It creates Deployment, Service, GatewayClass, and CRDs.

📦 Gateway API CRDs Installed

These should be automatically installed with the NGINX controller, but you can also apply them separately from the Kubernetes Gateway API GitHub repo:

```
kubectl apply -f
https://github.com/kubernetes-sigs/gateway-api/releases/down
load/v1.0.0/standard-install.yaml
```

✅ **Check:** Confirm CRDs like `gateways.gateway.networking.k8s.io`, `httproutes.gateway.networking.k8s.io` exist.

🧾 Existing Ingress Definitions That Need Migration

You should already have:

- Ingress manifests for services that expose HTTP(S) traffic
- Knowledge of how they behave: TLS, rewrites, headers, etc.

✅ **Action:** Identify and export these manifests for review and translation.

🔐 TLS Secrets (If You're Using HTTPS)

- If your apps currently use HTTPS, you'll need:
    - Existing TLS secrets
    - A plan to reuse them with Gateway listeners
    - Cert-manager setup (optional but recommended)

✅ **Action:** Ensure your certs are valid and available in the right namespace.

👥 Team Roles and RBAC Prepared

With Gateway API, roles should be separated:

- Platform team manages `GatewayClass` and `Gateway`
- Dev teams manage `HTTPRoute`

✅ **Action:** Define who manages what. Update RBAC to reflect these responsibilities.

🛠️ Tooling Adjustments (CI/CD, GitOps, etc.)

If you're using CI/CD pipelines:

- They might template or apply Ingress YAMLs
- You'll need to update those to handle Gateway API resources

✅ **Action:** Plan for pipeline and Helm chart updates.

🔍 Monitoring and Debugging Tools That Understand Gateway API

Your logging, metrics, and dashboards should:

- Track `Gateway` and `HTTPRoute` status
- Visualize traffic flow and errors
- Alert on failed route attachment or invalid configs

✅ **Action:** Integrate tools like:

- Prometheus (with custom metrics from controller)
- Grafana dashboards
- Gateway API status fields (`.status.conditions`)

🧪 A Safe Testing Environment

You should **never migrate directly in production**.

- Use a staging or test namespace
- Deploy services side-by-side (Ingress + Gateway)
- Compare responses, logs, and metrics

> ✅ **Action:** Prepare a parallel testing environment for each route or service being migrated.

🧩 Real-Time Task: Migrating `shop-app` Ingress to Gateway API using NGINX Gateway Controller on Kops

🧶 Context

You manage a production Kubernetes cluster using **Kops** on AWS. The cluster hosts a frontend application called `shop-app` running in the `production` namespace. Currently, traffic to `shop-app` is managed through a standard **Ingress** resource using the **classic NGINX Ingress Controller**.

The traffic is exposed under the hostname `shop.example.com`, with **TLS termination** handled by a certificate issued via `cert-manager`.

Your objective is to **migrate this Ingress setup to the Gateway API**, leveraging the **NGINX Gateway Controller**, which supports Kubernetes-native Gateway resources like `GatewayClass`, `Gateway`, and `HTTPRoute`.

The migration must:

- Maintain **TLS termination**
- Preserve existing routing behavior
- Avoid downtime
- Be implemented in a **production-safe** and **repeatable** manner

🛠️ Step-by-Step Implementation

✅ Step 1: Review the Existing Ingress (Current State)

Before migration, extract the existing Ingress definition.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: shop-app-ingress
  namespace: production
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  tls:
  - hosts:
    - shop.example.com
    secretName: shop-tls
  rules:
  - host: shop.example.com
```

```
  http:
    paths:
    - path: /
      pathType: Prefix
      backend:
        service:
          name: shop-app
          port:
            number: 80
```

🧠 **Note:** This Ingress provides HTTPS access to the `shop-app` service via NGINX Ingress Controller. The TLS certificate is stored in a secret named `shop-tls`.

⚙️ Step 2: Install NGINX Gateway Controller (Gateway API Edition)

You must use the version of NGINX that supports **Gateway API**, not the classic ingress controller.

Run the following command to install it:

```
kubectl apply -k "github.com/nginxinc/nginx-gateway-kubernetes/deploy/kubernetes/overlays/gateway-api?ref=v1.1.0"
```

📌 This installs:

- The NGINX Gateway controller Deployment
- Necessary CRDs (`Gateway`, `GatewayClass`, `HTTPRoute`, etc.)
- Service account, roles, and bindings

Verify the controller is running:

```
kubectl get pods -n nginx-gateway
```

🪪 Step 3: Create a GatewayClass

`GatewayClass` is the top-level abstraction that defines which controller handles which Gateway. In this case, you're assigning this to the NGINX Gateway Controller.

```
apiVersion: gateway.networking.k8s.io/v1
kind: GatewayClass
metadata:
  name: nginx
spec:
```

```
  controllerName: nginx.org/gateway-controller
```

Apply it:

```
kubectl apply -f gatewayclass.yaml
```

🌐 Step 4: Create the Gateway Resource

This defines the load balancer endpoint and TLS configuration. It listens on `shop.example.com` using port 443 (HTTPS).

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: shop-gateway
  namespace: production
spec:
  gatewayClassName: nginx
  listeners:
  - name: https
    protocol: HTTPS
```

```
    port: 443
    hostname: "shop.example.com"
    tls:
      certificateRefs:
      - kind: Secret
        name: shop-tls
    allowedRoutes:
      namespaces:
        from: Same
```

Explanation:

- `hostname`: Ensures that only traffic for `shop.example.com` is accepted.
- `tls.certificateRefs`: Refers to the same TLS secret used in the old Ingress.
- `allowedRoutes`: Ensures only HTTPRoutes from the `production` namespace can bind.

Apply the file:

```
kubectl apply -f gateway.yaml
```

🛣️ Step 5: Define the HTTPRoute

Now create the routing logic that replaces what was defined in the Ingress resource.

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: shop-route
  namespace: production
spec:
  parentRefs:
  - name: shop-gateway
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /
    backendRefs:
    - name: shop-app
      port: 80
```

Explanation:

- `parentRefs.name`: Links this route to the `shop-gateway`.
- `matches.path.value: /`: Captures all traffic like the previous Ingress.
- `backendRefs`: Sends traffic to the same service and port used by Ingress.

Apply the route:

```
kubectl apply -f httproute.yaml
```

🧪 Step 6: Validate the Migration

1. Verify Gateway & HTTPRoute status:

```
kubectl get gateway shop-gateway -n production

kubectl get httproute shop-route -n production
```

Check `.status.conditions[]` for `Accepted=True`, `Ready=True`.

2. DNS / TLS Testing:

If you use an external DNS (like Route53), point `shop.example.com` to the Gateway Service's external IP:

```
kubectl get svc -n nginx-gateway
```

Use `curl` to validate:

```
curl -k --resolve shop.example.com:443:<EXTERNAL-IP>
https://shop.example.com/
```

Expected:

- Valid TLS certificate
- Response from the `shop-app` service

🧹 Step 7: Cleanup the Ingress Resource

Once traffic is confirmed to be working with Gateway API:

```
kubectl delete ingress shop-app-ingress -n production
```

> 🎯 This finalizes the migration. All traffic is now served via Gateway API.

📌 Recap

You've just performed a **real-world migration** of a live application from **Ingress to Gateway API** with the following outcomes:

Component	Old (Ingress)	New (Gateway API)
Controller	NGINX Ingress	NGINX Gateway Controller
Resource	`Ingress`	`Gateway`, `HTTPRoute`
TLS Handling	`secretName` in Ingress	`certificateRefs` in Gateway
Routing Logic	`paths` in rules	`matches` and `backendRefs`
Namespace Scope	Flat	Explicit with `allowedRoutes`

✅ Real-Time Task: Migrating Multi-Path Ingress for `inventory-app` and `admin-panel` to Gateway API with NGINX Gateway Controller

🧶 Scenario

You manage a cluster where two frontend services — `inventory-app` and `admin-panel` — are exposed via a **single Ingress resource** on the domain `portal.example.com`.

The current structure:

- `https://portal.example.com/inventory` → routes to `inventory-app` service
- `https://portal.example.com/admin` → routes to `admin-panel` service

You are tasked with **migrating this multi-path Ingress to Gateway API**, while preserving:

- Path-based routing
- TLS termination
- Reuse of existing TLS cert (`portal-tls`)

- Clean separation between applications

🛠️ Step-by-Step Migration to Gateway API

✅ Step 1: Review Existing Ingress Setup

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: portal-ingress
  namespace: production
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /  # ensure root forwarding
spec:
  tls:
  - hosts:
    - portal.example.com
    secretName: portal-tls
  rules:
```

```
  - host: portal.example.com
    http:
      paths:
      - path: /inventory
        pathType: Prefix
        backend:
          service:
            name: inventory-app
            port:
              number: 80
      - path: /admin
        pathType: Prefix
        backend:
          service:
            name: admin-panel
            port:
              number: 80
```

⚙️ Step 2: Ensure NGINX Gateway Controller is Installed

Already installed from previous task:

```
kubectl get pods -n nginx-gateway
```

If not:

```
kubectl apply -k
"github.com/nginxinc/nginx-gateway-kubernetes/deploy/kuberne
tes/overlays/gateway-api?ref=v1.1.0"
```

🪪 Step 3: Define GatewayClass (Skip if already applied)

```
apiVersion: gateway.networking.k8s.io/v1
kind: GatewayClass
metadata:
  name: nginx
spec:
  controllerName: nginx.org/gateway-controller
```

Apply:

```
kubectl apply -f gatewayclass.yaml
```

🌐 Step 4: Create Gateway Resource (TLS Listener)

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: portal-gateway
  namespace: production
spec:
  gatewayClassName: nginx
  listeners:
  - name: https
    protocol: HTTPS
    port: 443
    hostname: "portal.example.com"
    tls:
      certificateRefs:
      - kind: Secret
        name: portal-tls
    allowedRoutes:
      namespaces:
```

```
        from: Same
```

Apply:

```
kubectl apply -f gateway.yaml
```

🛣️ Step 5: Create HTTPRoute for `/inventory`

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: inventory-route
  namespace: production
spec:
  parentRefs:
  - name: portal-gateway
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /inventory
```

```
    backendRefs:
    - name: inventory-app
      port: 80
```

Apply:

```
kubectl apply -f inventory-route.yaml
```

🛣️ Step 6: Create HTTPRoute for `/admin`

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: admin-route
  namespace: production
spec:
  parentRefs:
  - name: portal-gateway
  rules:
  - matches:
    - path:
```

```
        type: PathPrefix
        value: /admin
    backendRefs:
    - name: admin-panel
      port: 80
```

Apply:

```
kubectl apply -f admin-route.yaml
```

🧪 Step 7: Validate Routing

1. Verify resource status:

```
kubectl get gateway portal-gateway -n production
kubectl get httproute -n production
```

Check `.status.conditions[]` for `Accepted=True`.

2. Test each route:

```
curl -k --resolve portal.example.com:443:<GATEWAY-IP>
https://portal.example.com/inventory
curl -k --resolve portal.example.com:443:<GATEWAY-IP>
https://portal.example.com/admin
```

Expected result:

- Inventory and admin services respond independently
- TLS cert from `portal-tls` is valid

🧹 Step 8: Remove Old Ingress

```
kubectl delete ingress portal-ingress -n production
```

🧠 Additional Notes

- If either `/admin` or `/inventory` require authentication, you can use **Gateway filters** later.
- You can assign each HTTPRoute to a different dev team, improving multi-tenancy.
- This pattern is scalable: just add more routes and point them to new services.

✅ Task Summary

App	Route Path	HTTPRoute	Backend Service
Inventory App	`/inventory`	`inventory-route`	`inventory-app`
Admin Panel	`/admin`	`admin-route`	`admin-panel`

You have now successfully migrated a **multi-path TLS Ingress** to **modular, secure, and scalable Gateway API resources**, preserving all core behaviors while gaining future flexibility and team separation.

✅ Real-Time Task: Migrate `/api` and `/web` Paths to Gateway API with Header-Based Routing, TLS, and Delegation

🧶 Scenario

You manage a **Kops-based production cluster** hosting a monolithic Ingress that routes:

- `https://platform.example.com/api` → `api-service`
- `https://platform.example.com/web` → `frontend-service`

The cluster is multi-team:

- The **Platform team** controls TLS, DNS, and external gateway exposure.
- The **App teams** own their respective services and routing logic (`api-team`, `frontend-team`).

You must migrate this setup to **Gateway API using NGINX Gateway Controller**, implementing:

- TLS termination
- Clean **separation of concerns** using `allowedRoutes`

- Header-based routing on `/api` for versioning (`X-Api-Version: v1`)
- Team-specific ownership of `HTTPRoute`

🛠️ Step-by-Step Migration

✅ Step 1: Setup — Existing Ingress Definition (to be migrated)

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: platform-ingress
  namespace: platform
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  tls:
  - hosts:
    - platform.example.com
    secretName: platform-tls
  rules:
```

```
  - host: platform.example.com
    http:
      paths:
      - path: /api
        pathType: Prefix
        backend:
          service:
            name: api-service
            port:
              number: 80
      - path: /web
        pathType: Prefix
        backend:
          service:
            name: frontend-service
            port:
              number: 80
```

⚙️ Step 2: Install the NGINX Gateway Controller (if not already)

```
kubectl apply -k
"github.com/nginxinc/nginx-gateway-kubernetes/deploy/kuberne
tes/overlays/gateway-api?ref=v1.1.0"
```

🏛️ Step 3: Create the GatewayClass (Controller Binding)

```
apiVersion: gateway.networking.k8s.io/v1
kind: GatewayClass
metadata:
  name: nginx
spec:
  controllerName: nginx.org/gateway-controller
```

```
kubectl apply -f gatewayclass.yaml
```

🌐 Step 4: Create the Gateway (Managed by Platform Team)

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
```

```
  name: platform-gateway
  namespace: platform
spec:
  gatewayClassName: nginx
  listeners:
  - name: https
    protocol: HTTPS
    port: 443
    hostname: "platform.example.com"
    tls:
      certificateRefs:
      - kind: Secret
        name: platform-tls
    allowedRoutes:
      namespaces:
        from: Selector
        selector:
          matchLabels:
            route-access: allowed
```

```
kubectl apply -f gateway.yaml
```

🎯 Explanation:

- `allowedRoutes.from: Selector` gives control to selected namespaces.
- Only namespaces with label `route-access=allowed` can bind routes to this Gateway.
- TLS cert `platform-tls` is reused.

🧪 Step 5: Label Team Namespaces

```
kubectl label namespace api-team route-access=allowed

kubectl label namespace frontend-team route-access=allowed
```

🛣️ Step 6: HTTPRoute for `/api` (Owned by API Team) — With Header Routing

Namespace: `api-team`

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

kind: HTTPRoute

metadata:
```

```
  name: api-route
  namespace: api-team
spec:
  parentRefs:
  - name: platform-gateway
    namespace: platform
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /api
      headers:
      - name: X-Api-Version
        value: v1
    backendRefs:
    - name: api-service
      port: 80

kubectl apply -f api-route.yaml -n api-team
```

✅ Explanation:

- Traffic only routes to `api-service` **if** header `X-Api-Version: v1` is present.
- This enables future versioning (e.g., v2 routes, filters, etc.)

🖥️ Step 7: HTTPRoute for `/web` (Owned by Frontend Team)

Namespace: `frontend-team`

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: frontend-route
  namespace: frontend-team
spec:
  parentRefs:
  - name: platform-gateway
    namespace: platform
  rules:
  - matches:
    - path:
```

```
        type: PathPrefix
        value: /web
    backendRefs:
    - name: frontend-service
      port: 80
```

```
kubectl apply -f frontend-route.yaml -n frontend-team
```

🧪 Step 8: Validation

1. Confirm all resources are **Accepted and Ready**:

```
kubectl get gateway -n platform
kubectl get httproute -A
```

2. Test via curl:

```
curl -k --resolve platform.example.com:443:<LB-IP>
https://platform.example.com/api \
  -H "X-Api-Version: v1"
```

```
curl -k --resolve platform.example.com:443:<LB-IP>
https://platform.example.com/web
```

> You should receive valid responses from both apps.
> Header-based routing should reject requests that don't match the expected version header.

🧹 Step 9: Cleanup Old Ingress

Once validated:

```
kubectl delete ingress platform-ingress -n platform
```

✅ Summary: What You Accomplished

Feature	Implementation
TLS Termination	Managed by Gateway on port 443
Path Routing	`/api` and `/web` via HTTPRoute
Header Routing	`X-Api-Version: v1` on `/api`
Namespace Delegation	`allowedRoutes` via label selector
Team Ownership	Separate routes per namespace
NGINX Controller Used	NGINX Gateway Controller

✅ Real-Time Task: Blue-Green Deployment for `payment-service` Using Gateway API Weighted Routing

🧶 Scenario

You're managing the `payment-service` which currently runs version `v1` in production under:

`https://pay.example.com/`

You're preparing to release a new version `v2`, deployed as a separate service `payment-service-v2`. The goal is to **migrate traffic from v1 to v2 gradually** without downtime.

Your current Ingress exposes `payment-service-v1`. You must now:

- Migrate this route to Gateway API
- Use **weighted traffic splitting**
- Route 90% to `v1`, 10% to `v2`
- Increase traffic over time until full migration
- Rollback if needed

🛠️ Full Implementation on Kops + NGINX Gateway Controller

✅ Step 1: Current Ingress (Reference Only)

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: payment-ingress
  namespace: production
spec:
  tls:
  - hosts:
    - pay.example.com
    secretName: pay-tls
  rules:
  - host: pay.example.com
    http:
      paths:
      - path: /
```

```
        pathType: Prefix
        backend:
          service:
            name: payment-service-v1
            port:
              number: 80
```

✅ Step 2: Create Gateway and TLS Setup

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: payment-gateway
  namespace: production
spec:
  gatewayClassName: nginx
  listeners:
  - name: https
    protocol: HTTPS
    port: 443
```

```
    hostname: "pay.example.com"
    tls:
      certificateRefs:
      - kind: Secret
        name: pay-tls
    allowedRoutes:
      namespaces:
        from: Same
```

```
kubectl apply -f payment-gateway.yaml
```

✅ Step 3: Deploy v2 Alongside v1

You should already have:

```
kubectl get svc payment-service-v1 -n production
```

Now deploy:

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

```
  name: payment-service-v2
  namespace: production
spec:
  selector:
    app: payment-service
    version: v2
  ports:
  - port: 80
    targetPort: 8080
```

And the backing deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: payment-v2
  namespace: production
spec:
  replicas: 2
  selector:
```

```
    matchLabels:
      app: payment-service
      version: v2
  template:
    metadata:
      labels:
        app: payment-service
        version: v2
    spec:
      containers:
      - name: payment
        image: your-repo/payment:v2
        ports:
        - containerPort: 8080
```

Apply it:

```
kubectl apply -f payment-v2.yaml
```

✅ Step 4: Create HTTPRoute With Weighted Routing

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: payment-route
  namespace: production
spec:
  parentRefs:
  - name: payment-gateway
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /
    backendRefs:
    - name: payment-service-v1
      port: 80
      weight: 90
    - name: payment-service-v2
```

```
      port: 80
      weight: 10
```

Apply:

```
kubectl apply -f payment-route.yaml
```

🎯 Now:

- 90% of traffic goes to `payment-service-v1`
- 10% of traffic goes to `payment-service-v2`

✅ Step 5: Validate Rollout

1. Test in browser or curl repeatedly:

```
curl -k --resolve pay.example.com:443:<LB-IP>
https://pay.example.com/
```

2. Log both services to confirm traffic is split.

```
kubectl logs -l app=payment-service,version=v1 -n production
kubectl logs -l app=payment-service,version=v2 -n production
```

3. Adjust weights when confident:

```
backendRefs:
- name: payment-service-v1
  port: 80
  weight: 50
- name: payment-service-v2
  port: 80
  weight: 50
```

Eventually:

```
- name: payment-service-v1
  port: 80
  weight: 0
- name: payment-service-v2
  port: 80
  weight: 100
```

Apply each change with:

```
kubectl apply -f payment-route.yaml
```

🚨 Step 6: Rollback (if needed)

Simply restore weight to v1:

```
- name: payment-service-v1
  port: 80
  weight: 100
- name: payment-service-v2
  port: 80
  weight: 0
```

You can also redeploy previous version under v2 service if needed.

✅ Step 7: Remove Ingress

After complete migration and full traffic on Gateway API:

```
kubectl delete ingress payment-ingress -n production
```

✅ Real-Time Task: Migrate Ingress with Multi-Tenant Subdomain Routing to Gateway API (Isolated Namespaces + TLS)

🧶 Scenario

Your company operates a **multi-tenant SaaS platform**. Each tenant gets a subdomain like:

- `tenant1.example.com`
- `tenant2.example.com`

Previously, a single **Ingress** routed based on hostname and forwarded traffic to services in each tenant's namespace.

You now need to migrate this to **Gateway API**, with:

- Subdomain-based routing (`*.example.com`)
- TLS termination for each tenant
- Each tenant managing its own routing logic (in isolated namespaces)
- Platform team owning and managing the Gateway (no tenant can modify it)
- DNS and TLS certs already handled via wildcard `*.example.com` cert

You will implement:

- One `Gateway` (owned by platform team)
- Tenant-specific `HTTPRoute` (per-namespace, isolated)
- `allowedRoutes.from: Selector` to strictly allow selected tenant namespaces
- TLS wildcard cert for `*.example.com`

🛠️ Step-by-Step Implementation

✅ Step 1: Pre-Migration Ingress Setup (Reference)

Namespace: `platform`

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: multi-tenant-ingress
  namespace: platform
spec:
  tls:
```

```
  - hosts:
    - "*.example.com"
    secretName: wildcard-cert
  rules:
  - host: tenant1.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: tenant1-web
            port:
              number: 80
  - host: tenant2.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
```

```
        backend:
          service:
            name: tenant2-web
            port:
              number: 80
```

✅ Step 2: Create GatewayClass (Controller Binding)

```
apiVersion: gateway.networking.k8s.io/v1
kind: GatewayClass
metadata:
  name: nginx
spec:
  controllerName: nginx.org/gateway-controller
```

```
kubectl apply -f gatewayclass.yaml
```

✅ Step 3: Create the Shared Gateway (Owned by Platform)

Namespace: `platform`

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: tenant-gateway
  namespace: platform
spec:
  gatewayClassName: nginx
  listeners:
  - name: https
    protocol: HTTPS
    port: 443
    hostname: "*.example.com"
    tls:
      certificateRefs:
      - name: wildcard-cert
        kind: Secret
    allowedRoutes:
```

```
    namespaces:
      from: Selector
      selector:
        matchLabels:
          allow-gateway-bind: "true"
```

✅ This allows only tenant namespaces with label `allow-gateway-bind=true` to bind routes.

```
kubectl apply -f tenant-gateway.yaml
```

✅ Step 4: Setup Tenant 1 Namespace and App

```
kubectl create ns tenant1
kubectl label ns tenant1 allow-gateway-bind=true
```

Then deploy the tenant's service:

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

```
  name: tenant1-web
  namespace: tenant1
spec:
  selector:
    app: tenant1-web
  ports:
  - port: 80
    targetPort: 8080
```

And its Deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tenant1-web
  namespace: tenant1
spec:
  replicas: 2
  selector:
    matchLabels:
```

```
      app: tenant1-web
  template:
    metadata:
      labels:
        app: tenant1-web
    spec:
      containers:
      - name: web
        image: your-repo/tenant1-app:v1
        ports:
        - containerPort: 8080
```

✅ Step 5: Tenant 1 HTTPRoute (Bound to Gateway)

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: tenant1-route
  namespace: tenant1
spec:
```

```
  parentRefs:
  - name: tenant-gateway
    namespace: platform
  hostnames:
  - "tenant1.example.com"
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /
    backendRefs:
    - name: tenant1-web
      port: 80
```

```
kubectl apply -f tenant1-route.yaml
```

🎯 This binds only `tenant1.example.com` to `tenant1-web` — **tenant 1 owns their route, platform team owns the gateway**.

✅ Step 6: Repeat for Tenant 2

```
kubectl create ns tenant2

kubectl label ns tenant2 allow-gateway-bind=true
```

Create `tenant2-web` service and deployment (similar to tenant1), then:

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: tenant2-route
  namespace: tenant2
spec:
  parentRefs:
  - name: tenant-gateway
    namespace: platform
  hostnames:
  - "tenant2.example.com"
  rules:
  - matches:
    - path:
```

```
        type: PathPrefix
        value: /
    backendRefs:
    - name: tenant2-web
      port: 80
```

```
kubectl apply -f tenant2-route.yaml
```

🧪 Step 7: Validate

```
curl -k --resolve tenant1.example.com:443:<LB-IP>
https://tenant1.example.com/
```

```
curl -k --resolve tenant2.example.com:443:<LB-IP>
https://tenant2.example.com/
```

Check route status:

```
kubectl get httproute -A
```

```
kubectl describe gateway tenant-gateway -n platform
```

🧹 Step 8: Decommission Ingress

```
kubectl delete ingress multi-tenant-ingress -n platform
```

✅ Real-Time Task: Migrate Authenticated Internal Apps to Gateway API with External OIDC Authentication Layer

🧶 Scenario

You have an internal dashboard service (`internal-dashboard`) exposed only to employees. Today, it's behind a basic Ingress protected by IP whitelisting. This approach is insufficient for:

- Remote employees
- Identity-based access
- Auditing requirements

Your goal:

- Migrate the `internal-dashboard` ingress to **Gateway API**
- Use an **OIDC-compliant identity provider (e.g., Auth0, Google, Okta)**
- Authenticate users using OAuth2 at the gateway layer (using NGINX Gateway Controller + external auth)

- Allow access **only to users in a specific email domain**
- Ensure secure TLS via Gateway API
- Use an external auth service for token validation

This pattern gives you **Zero Trust, identity-aware access** — without rewriting the app.

🛠️ Implementation: Gateway API + External Auth + OIDC

✅ Step 1: Assumptions

- Domain: `dashboard.example.com`
- OIDC provider: Auth0 (could be any)
- External Auth service: `oauth2-proxy`
- TLS cert is already available as secret `internal-tls` in namespace `platform`
- Application is already deployed as `internal-dashboard` in `internal` namespace

✅ Step 2: Deploy OAuth2 Proxy (as External Auth)

Create the proxy in the `security` namespace:

```
kubectl create ns security
```

Sample Deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: oauth2-proxy
  namespace: security
spec:
  replicas: 1
  selector:
    matchLabels:
      app: oauth2-proxy
  template:
    metadata:
      labels:
        app: oauth2-proxy
```

```
    spec:
      containers:
      - name: oauth2-proxy
        image: quay.io/oauth2-proxy/oauth2-proxy:v7.6.0
        args:
        - --provider=oidc
        - --oidc-issuer-url=https://<AUTH0-DOMAIN>/
        - --client-id=<CLIENT-ID>
        - --client-secret=<CLIENT-SECRET>
        - --cookie-secret=<COOKIE-SECRET>
        - --email-domain=example.com
        - --upstream=http://127.0.0.1:4180/
        - --http-address=0.0.0.0:4180
        ports:
        - containerPort: 4180
```

And expose it:

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

```
metadata:
  name: oauth2-proxy
  namespace: security
spec:
  selector:
    app: oauth2-proxy
  ports:
  - port: 80
    targetPort: 4180
```

✅ Step 3: Create Gateway (HTTPS Listener)

In namespace `platform`:

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: internal-gateway
  namespace: platform
spec:
  gatewayClassName: nginx
```

```
  listeners:
  - name: https
    protocol: HTTPS
    port: 443
    hostname: "dashboard.example.com"
    tls:
      certificateRefs:
      - name: internal-tls
    allowedRoutes:
      namespaces:
        from: Same
```

✅ Step 4: Deploy the Internal App (if not already)

Namespace: `internal`

```
apiVersion: v1
kind: Service
metadata:
  name: internal-dashboard
  namespace: internal
```

```
spec:
  selector:
    app: dashboard
  ports:
  - port: 80
    targetPort: 8080
```

✅ Step 5: HTTPRoute With External Authentication Filter

Namespace: `internal`

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: internal-dashboard-route
  namespace: internal
spec:
  parentRefs:
  - name: internal-gateway
    namespace: platform
  hostnames:
```

```
  - "dashboard.example.com"
  rules:
  - filters:
    - type: RequestAuthentication
      requestAuthentication:
        extensionRef:
          group: config.gateway.nginx.org
          kind: ExternalAuth
          name: oauth2-auth
    - type: RequestHeaderModifier
      requestHeaderModifier:
        add:
        - name: X-Forwarded-User
          value: "authenticated"
    backendRefs:
    - name: internal-dashboard
      port: 80
```

✅ Step 6: Define the External Auth Policy (CustomResource)

This is NGINX-specific and handled via their CRD:

```
apiVersion: config.gateway.nginx.org/v1alpha1
kind: ExternalAuth
metadata:
  name: oauth2-auth
  namespace: internal
spec:
  request:
    service:
      name: oauth2-proxy
      port: 80
    pathPrefix: /
  response:
    allowedStatusCodes: [200]
    headers:
      - name: X-Auth-Request-Email
      - name: X-Auth-Request-User
```

🎯 This tells NGINX Gateway to:

- Forward request to `oauth2-proxy`
- Allow only if response status is 200
- Pass identity headers to app

🧪 Step 7: Validate

Open browser:
`https://dashboard.example.com`

1. You'll be redirected to Auth0 login.
2. On success, traffic reaches your dashboard.

Confirm user identity headers in app logs:

```
X-Auth-Request-Email: user@example.com
```

🔁 Step 8: Migration Complete — Remove Ingress

```
kubectl delete ingress internal-dashboard-ingress -n internal
```

✅ Key Outcomes

Security Feature	Implementation via Gateway API
OIDC Authentication	`oauth2-proxy` external filter
Route-level Auth	`RequestAuthentication` filter
TLS	Full termination in Gateway
Identity Isolation	Only `example.com` domain users allowed
Gateway Ownership	Platform team retains listener control
Least Privilege	Apps unaware of auth — Zero Trust pattern

🧰 Troubleshooting & Debugging Gateway API (NGINX Controller)

🔎 1. Validate Gateway API Resources

Check GatewayClass

```
kubectl get gatewayclass
```

```
kubectl describe gatewayclass nginx
```

> ✅ Make sure the `controllerName` matches `nginx.org/gateway-controller`.

Check Gateway

```
kubectl get gateways -A
```

```
kubectl describe gateway <gateway-name> -n <namespace>
```

> 🔍 Look for `Address:` field — it should contain a LoadBalancer IP/hostname.
>
> If it's missing, the controller hasn't provisioned it yet.

📦 2. Verify Gateway Address Assignment (LoadBalancer/IP)

```
kubectl get svc -n <nginx-gateway-namespace>
```

Look for the `LoadBalancer` service, such as:

```
nginx-gateway   LoadBalancer   34.120.XXX.XXX
443:32443/TCP
```

If not showing:

- Check if cloud provider integration is configured (important for **Kops**)
- Check controller logs

📡 3. Validate HTTPRoute Binding

```
kubectl get httproute -A
```

```
kubectl describe httproute <route-name> -n <namespace>
```

Look for:

- **`ParentRefs:`** — should point to the correct Gateway

- **`Accepted: True`** — under `status.parents.conditions`

❌ If `Accepted=False`, reason will be shown (e.g., `RefNotPermitted`, `NoMatchingParent`)

🔐 4. TLS Troubleshooting

Check TLS Certificate Secret

```
kubectl get secret <secret-name> -n <namespace>
```

```
kubectl describe secret <secret-name> -n <namespace>
```

- Make sure it's type `kubernetes.io/tls`
- Validate base64 content manually if needed

Confirm Listener TLS Binding

```
kubectl get gateway <name> -n <namespace> -o yaml | grep -A10 listeners
```

Look for:

- `protocol: HTTPS`

- `certificateRefs`
- `hostname: "*.example.com"` (or exact match)

🔐 5. Debug External Auth Filters

If using `ExternalAuth` (e.g., with `oauth2-proxy`):

Check Logs of Auth Proxy

```
kubectl logs -l app=oauth2-proxy -n security
```

Look for:

- 401/403 errors → may indicate failed OIDC auth
- Redirect loops → often cookie or upstream misconfig
- Invalid client ID/secret → misconfigured OIDC

Check Route Filter Status

```
kubectl describe httproute <name> -n <namespace>
```

Check `filters` section → ensure filter is recognized.
If it's not, you may be missing the CRD or plugin is not installed.

🔀 6. Debug Request Routing (Live Traffic)

Use `curl` with Host Header

```
curl -v https://<LB-IP>/ -H "Host: your.example.com"
```

Or resolve domain:

```
curl -v --resolve your.example.com:443:<LB-IP>
https://your.example.com/
```

> 🔍 Use this to test domain routing without needing real DNS set up

📋 7. Events: First Place to Check for Failures

```
kubectl get events -A | grep -Ei
'gateway|httproute|error|fail'
```

This will reveal most binding failures, listener issues, or cert misconfigs.

📄 8. Controller Logs (NGINX)

```
kubectl logs -l app=nginx-gateway -n <namespace>
```

Use `-f` for live view:

```
kubectl logs -f deployment/nginx-gateway -n <namespace>
```

> 🔥 Crucial for seeing why a Gateway or Route failed to bind or why certs aren't being picked up.

🧼 9. Clean up & Reapply for Debugging

Sometimes resources get stuck in `Pending` or `Invalid`.

Safe order to delete:

```
kubectl delete httproute <name> -n <ns>

kubectl delete gateway <name> -n <ns>

kubectl delete gatewayclass nginx
```

Then reapply with:

```
kubectl apply -f gateway.yaml

kubectl apply -f httproute.yaml
```

🧪 10. Tools & Tips

Use **`kubectl explain`** to explore Gateway API fields:

```
kubectl explain gateway.networking.k8s.io/v1.HTTPRoute
```

- Use **`kubectl api-resources | grep gateway`** to verify CRDs are installed.

Visualize resource status:

```
kubectl get httproute -A -o wide
```

- Watch all gateway API resources:

  ```
  watch "kubectl get gatewayclass,gateway,httproute -A"
  ```

📘 Wrapping Up

Migrating from the traditional Ingress resource to the modern Gateway API is more than just a switch in APIs — it’s a step toward scalable, secure, and production-grade traffic management in Kubernetes. With improved separation of responsibilities, richer routing capabilities, and better extensibility, Gateway API enables platform teams and application teams to collaborate efficiently while reducing operational risks.

In this doc, we explored:

- Why and when to migrate from Ingress to Gateway API
- Precautions and pre-requisites for a smooth migration
- Multiple real-world scenario-based tasks with NGINX Gateway Controller
- Troubleshooting insights and production-grade best practices

> 💡 Whether you're building for high-scale microservices or zero-downtime blue-green deployments, Gateway API is the future of traffic control in Kubernetes.

🚀 What's Next?

Stay tuned! I'll be sharing more deep-dive Kubernetes docs like:

- Canary Deployments with Gateway API
- Multi-tenant Gateway setups
- Custom filters and security policies
- GitOps-based Gateway management

  ...and much more tailored to **DevOps, SecOps, and Platform Engineering roles**.

Thanks for reading and supporting this series.

Let's build infrastructure that's not just functional — but future-ready. 👨‍💻⚙️🌐

– Fenil Gajjar

"Real-world engineering beats theoretical perfection."