# Ingress & Cluster Issuer

## 1. Ingress as an API Object

An Ingress is a **Kubernetes-native declarative configuration** that defines routing rules for inbound HTTP(S) traffic. It sits at **layer 7 (HTTP/HTTPS)** of the OSI model and leverages the Kubernetes API to:

- Define **host-based** and **path-based** routing rules
- Delegate **TLS termination**
- Coordinate with external software (Ingress Controller) to materialize the routing logic

However, **Ingress itself is not a traffic handler**. It merely declares how traffic **should be routed**. The actual routing logic is implemented by the **Ingress Controller**.

---

## 2. Ingress Controller

The Ingress Controller is a **daemon** or **deployment** inside the Kubernetes cluster, responsible for:

- Watching for changes in Ingress, Service, and Secret resources
- Dynamically reconfiguring its reverse proxy (e.g., NGINX, Traefik)
- Terminating TLS (if enabled)
- Enforcing route rewrites, redirects, security policies, timeouts, rate limiting, etc.

When using nginx-ingress-controller, the controller parses the Ingress spec and modifies the internal **nginx.conf** using a templating engine. All traffic routing happens inside this dynamic configuration.

---

## 3. Anatomy of an Ingress Resource (Dissected)

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: sample-ingress
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
    - host: www.example.com
```

```
   http:
     paths:
       - path: /app
         pathType: Prefix
         backend:
           service:
             name: app-service
             port:
               number: 8080
tls:
  - hosts:
      - www.example.com
    secretName: example-tls
```

**Key Segments:**

- metadata.annotations: Controller-specific configurations. These annotations are **not validated by the Kubernetes API server**; they are **interpreted by the Ingress Controller**.
- ingressClassName: Matches the controller deployment watching for this ingress. If misconfigured, the Ingress resource is ignored.
- rules: Define routing logic. It supports:
  - Host-based routing
  - Path-based routing
- tls: Specifies a Kubernetes Secret containing tls.crt and tls.key.

This YAML doesn't serve HTTPS unless:

1. The secret example-tls exists
2. The Ingress Controller is correctly configured with TLS support

---

### 4. IngressController Internal Operation Flow

Upon creation of an Ingress resource:

- The Ingress Controller watches for Ingress and Service events
- It builds a **routing tree** (host → path → service) in memory
- If TLS is enabled:
  - It checks if the TLS secret exists
  - It loads the certificate and key dynamically

- If rewrite or redirect annotations exist:
    - It alters the reverse proxy rules accordingly
- It regenerates the proxy configuration (e.g., NGINX reloads its configuration)

This reload is **graceful**, done via signal-based triggers (nginx -s reload) and without dropping active connections.

---

**5. Network Flow With Ingress and TLS**

**Example: TLS-enabled Ingress using nginx-ingress-controller**

1. DNS resolves www.example.com to the ELB/IP address of the Ingress Controller
2. Browser initiates TCP + TLS handshake on port 443
3. nginx-ingress-controller receives connection
4. It presents the TLS certificate loaded from the secret
5. If the handshake is successful, HTTPS session begins
6. The Controller decrypts the data and routes the HTTP request internally to the Kubernetes Service defined in the rule
7. Internal traffic (from controller to pod) is typically **unencrypted HTTP**, unless mTLS is enforced manually

---

# CLUSTERISSUER

---

**1. cert-manager: TLS Lifecycle Automation**

cert-manager is a Kubernetes-native controller that automates the management of TLS certificates.

It handles:

- Automated issuance
- Renewal before expiry
- Storage in Kubernetes Secrets
- Integration with Ingress for HTTPS

It uses the **ACME protocol** (Automated Certificate Management Environment) to talk to CAs like Let's Encrypt.

---

**2. What is a ClusterIssuer?**

A ClusterIssuer is a **cluster-scoped resource** that defines how cert-manager should communicate with a CA to:

- Prove ownership of a domain
- Request a signed TLS certificate
- Store that certificate as a Kubernetes TLS secret

It differs from an Issuer only in **scope**: Issuers are namespaced, while ClusterIssuers are cluster-wide.

---

**3. Anatomy of a ClusterIssuer**

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
spec:
  acme:
    server: https://acme-v02.api.letsencrypt.org/directory
    email: admin@example.com
    privateKeySecretRef:
      name: letsencrypt-prod-account-key
    solvers:
      - http01:
```

```
      ingress:
        class: nginx
```

**Key Fields:**

- server: ACME directory URL. Let’s Encrypt offers staging and production endpoints.
- email: Contact address for expiration notices or abuse reports.
- privateKeySecretRef: Used to store the **ACME account key** for authenticating with Let’s Encrypt.
- solvers: Defines how the domain ownership challenge will be solved.
  - http01: Proves control by serving a token at a specific URL under .well-known/acme-challenge/
  - The ingress.class refers to the controller (usually nginx) used to create the temporary routing rule

---

**4. Challenge Flow (ACME HTTP-01, Full Lifecycle)**

When an Ingress with TLS and a cert-manager.io/cluster-issuer annotation is created:

1. cert-manager detects the Ingress and determines it needs a certificate for www.example.com
2. cert-manager creates a Certificate resource internally
3. cert-manager registers an ACME account using the ClusterIssuer credentials
4. cert-manager sends a request to the ACME server for a certificate
5. The ACME server (e.g., Let’s Encrypt) responds:
   - Prove ownership of www.example.com by placing a file with a specific token at:

http://www.example.com/.well-known/acme-challenge/<TOKEN>

6. cert-manager creates:
   - A temporary Ingress that routes that token to a dedicated **Challenge solver pod**
   - The pod is mounted with a volume containing the challenge response
7. Let’s Encrypt makes an HTTP GET request to the token URL
8. If the response is correct, the challenge passes
9. Let’s Encrypt signs and returns the certificate
10. cert-manager stores the certificate and key in a Secret (tls.crt, tls.key)

11. The production Ingress now reads the certificate from this secret

---

**5. Secret Creation and Use in Ingress**

Once cert-manager stores the certificate in the TLS secret, the Ingress Controller uses it during TLS handshake.

The Ingress TLS block must reference this secret:

```
tls:
  - hosts:
      - www.example.com
    secretName: example-com-tls
```

If the secret is missing or invalid:

- The Ingress Controller serves an **invalid or default certificate**
- TLS handshakes will fail due to mismatch or untrusted certs

---

**6. Lifecycle of Certificates**

cert-manager constantly watches Certificate resources and:

- Automatically renews certificates 30 days before expiration
- If renewal fails, logs the reason and sends events
- New cert is written to the same secret
- Ingress Controller automatically uses the new certificate (reloads it dynamically)

No human involvement is needed after initial setup.

---

**7. Interactions Among All Components**

| Component | Role |
|---|---|
| ClusterIssuer | Defines CA details and challenge-solving strategy |
| cert-manager | Automates cert lifecycle, solves challenges, writes secrets |
| Ingress | Declares TLS requirement and backend routing |
| Ingress Controller | Implements the Ingress rules and TLS termination |
| ACME Server (e.g. LE) | Validates domain control, signs certificates |
| Kubernetes Secret | Stores the final TLS certificate and key |
| DNS | Must point to the public IP of the Ingress Controller |

**Conclusion**

- **Ingress** is the routing declaration layer. It decides how to route external traffic to internal services.
- **Ingress Controller** is the runtime implementation. It reads Ingress and Service definitions and configures reverse proxy rules accordingly.
- **ClusterIssuer** defines how cert-manager should issue TLS certificates, including the use of Let’s Encrypt via ACME.
- **cert-manager** is the engine that automates challenge-solving, cert issuance, renewal, and secret management.
- These components work together to establish **fully automated HTTPS at the Kubernetes edge**, without any manual key handling or certificate tracking.