# HTTPS Forwarding in Kubernetes

how a https request forward from web to pod in k8s explaining step by step and give flow chart diagram

To understand how an HTTPS request is forwarded from a web client to a pod in Kubernetes, let's go through the step-by-step process. We'll assume a typical setup using an Ingress controller for handling HTTPS traffic. Here's an overview of the steps:

1. **Client Initiates HTTPS Request:**
   - A user initiates an HTTPS request from their browser or any other client to a specific URL.
2. **DNS Resolution:**
   - The client's browser sends a DNS query to resolve the domain name to an IP address.
3. **Request Reaches Load Balancer:**
   - The resolved IP address typically points to a Load Balancer (e.g., AWS ELB) configured to handle incoming traffic for the Kubernetes cluster.
   - The Load Balancer accepts the HTTPS request on port 443.
4. **Forward to Ingress Controller:**
   - The Load Balancer forwards the HTTPS request to the Ingress Controller running in the Kubernetes cluster.
   - The Ingress Controller is responsible for handling and routing HTTP/HTTPS traffic to the appropriate services within the cluster.
5. **TLS Termination:**
   - The Ingress Controller terminates the TLS (HTTPS) connection, decrypting the traffic.
   - The Ingress Controller can be configured with TLS certificates to handle secure connections.
6. **Route to Service:**
   - The Ingress Controller uses the Ingress resource rules to determine the appropriate Kubernetes Service to route the traffic to.
   - The Ingress resource defines rules based on the host and path of the incoming request.
7. **Forward to Pod:**
   - The Kubernetes Service is responsible for selecting the appropriate pod(s) to handle the request.
   - The Service routes the traffic to one of the pods, which may involve further load balancing.
8. **Pod Handles Request:**
   - The selected pod processes the incoming request, executes the necessary application logic, and generates a response.
   - The response is sent back through the same path: from the pod to the Service, to the Ingress Controller, to the Load Balancer, and finally back to the client.

Here's a flowchart to illustrate this process:

```
mermaid
```

```
flowchart TD
    A[Client] -->|HTTPS Request| B[DNS Server]
    B -->|Resolve Domain| C[Load Balancer]
    C -->|Forward HTTPS| D[Ingress Controller]
    D -->|TLS Termination & Routing| E[Kubernetes Service]
    E -->|Forward to Pod| F[Pod]
    F -->|Process Request & Generate Response| E
    E --> D
    D --> C
    C --> A
```

### Key Components:

- **Client:** The end user initiating the HTTPS request.
- **DNS Server:** Resolves the domain name to the IP address of the Load Balancer.
- **Load Balancer:** Distributes incoming traffic to the Ingress Controller.
- **Ingress Controller:** Manages HTTPS termination and routing rules to direct traffic to the appropriate services.
- **Kubernetes Service:** Provides a stable endpoint and load balances traffic to the pods.
- **Pod:** The containerized application instance that processes the request.

This diagram and explanation provide a high-level overview of how an HTTPS request is forwarded from a web client to a pod in a Kubernetes cluster.

give an reak time example of these Key Components and explainbreifly

Let's go through a real-time example of these key components in the context of an e-commerce application deployed in a Kubernetes cluster. The domain for the e-commerce application is `**shop.example.com**`.

### 1. Client Initiates HTTPS Request

- **Client:** Alice, a customer, opens her web browser and navigates to `**https://shop.example.com**`.

### 2. DNS Resolution

- **DNS Server:** Alice's browser sends a DNS query to resolve `**shop.example.com**` to an IP address. The DNS server responds with the IP address of the Load Balancer.

### 3. Request Reaches Load Balancer

- **Load Balancer:** The IP address points to an AWS Elastic Load Balancer (ELB) that is set up to handle incoming traffic for the Kubernetes cluster. The ELB accepts Alice's HTTPS request on port 443.

### 4. Forward to Ingress Controller

- **Ingress Controller:** The Load Balancer forwards Alice's HTTPS request to the Ingress Controller running in the Kubernetes cluster. The Ingress Controller, such as NGINX Ingress Controller, is configured to manage and route traffic based on defined Ingress resources.

### 5. TLS Termination

- **TLS Termination:** The Ingress Controller terminates the TLS connection, decrypting the HTTPS traffic. It uses a TLS certificate (e.g., from Let's Encrypt) configured for `shop.example.com`.

### 6. Route to Service

- **Ingress Resource:** The Ingress Controller checks its Ingress rules, which are configured to route traffic based on the host (`shop.example.com`) and path (e.g., `/products`, `/cart`).
- **Kubernetes Service:** According to the Ingress rules, the Ingress Controller forwards the request to a Kubernetes Service named `web-service`. This service provides a stable endpoint and load balances the traffic to the appropriate pods.

### 7. Forward to Pod

- **Service to Pod:** The `web-service` selects one of the available pods running the e-commerce application. This selection is typically done using a round-robin or another load-balancing algorithm.

### 8. Pod Handles Request

- **Pod:** The selected pod (e.g., `web-pod-1`) processes Alice's request, executes the application logic (such as querying the product database), and generates an HTTP response.
- **Response Path:** The response is sent back through the same path: from the pod to the `web-service`, then to the Ingress Controller, the Load Balancer, and finally to Alice's browser.

### Real-time Example Flowchart

```mermaid
flowchart TD
    A[Alice's Browser] -->|HTTPS Request| B[DNS Server]
    B -->|Resolve shop.example.com| C[AWS ELB]
    C -->|Forward HTTPS| D[NGINX Ingress Controller]
    D -->|TLS Termination & Routing| E[web-service]
    E -->|Forward to web-pod-1| F[web-pod-1]
    F -->|Process Request & Generate Response| E
    E --> D
    D --> C
    C --> A
```

### Brief Explanation of Key Components:

1. **Client (Alice's Browser):**
   - Initiates the HTTPS request to access the e-commerce website.
2. **DNS Server:**

   - Resolves the domain name `shop.example.com` to the IP address of the Load Balancer.
3. **Load Balancer (AWS ELB):**
   - Accepts incoming HTTPS requests and forwards them to the Ingress Controller in the Kubernetes cluster.
4. **Ingress Controller (NGINX Ingress Controller):**
   - Handles TLS termination, decrypting the HTTPS traffic, and routes the request based on defined Ingress rules.
5. **Kubernetes Service (web-service):**
   - Acts as an intermediary to forward the request to the appropriate pod, providing load balancing.
6. **Pod (web-pod-1):**
   - Processes the request, executes application logic, and generates a response to be sent back to the client.

In this example, the entire flow demonstrates how a user's HTTPS request is processed and routed within a Kubernetes cluster, ensuring secure and efficient handling of web traffic.

give the ingress controler file and service file and explanation

Sure, let's go through the example configuration files for an Ingress Controller and a Service in a Kubernetes cluster for an e-commerce application.

### Ingress Controller Configuration

First, you need to ensure that an Ingress Controller is installed in your cluster. For this example, we'll use the NGINX Ingress Controller. You can install it using the following command:

```sh
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/main/deploy/static/provider/cloud/deploy.yaml
```

### Ingress Resource Configuration

Here's an example Ingress resource file for routing traffic to a `web-service`:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: shop-ingress
  namespace: default
  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: web-service
            port:
              number: 80
```

## Explanation of the Ingress Resource File

- **apiVersion, kind, metadata:** Standard Kubernetes fields specifying the API version, resource type, and metadata (name and namespace).
- **annotations:** Additional configuration options for the Ingress controller. In this case, `nginx.ingress.kubernetes.io/rewrite-target` is used to rewrite the URL path.
- **tls:** Specifies the TLS settings. The `hosts` field lists the domain(s) covered by this Ingress, and `secretName` refers to the Kubernetes Secret containing the TLS certificate.
- **rules:** Specifies the routing rules. In this example, traffic to `shop.example.com` is forwarded to the `web-service` on port 80.

## Kubernetes Service Configuration

Here's an example Service file for the `web-service`:

yaml

```
apiVersion: v1
kind: Service
metadata:
  name: web-service
  namespace: default
spec:
  selector:
    app: web-app
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
  type: ClusterIP
```

## Explanation of the Service File

- **apiVersion, kind, metadata:** Standard Kubernetes fields specifying the API version, resource type, and metadata (name and namespace).
- **selector:** Defines which pods the service targets. The service will route traffic to pods with the label `app=web-app`.
- **ports:** Specifies the ports the service exposes and the corresponding target ports on the pods. Here, port 80 is exposed by the service, and traffic is forwarded to port 8080 on the pods.
- **type:** Specifies the type of service. `ClusterIP` means the service is only accessible within the cluster. For external access, other types like `NodePort` or `LoadBalancer` can be used.

### Deployment Configuration (Optional)

For completeness, here's an example Deployment file for the `web-app` pods:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-app-deployment
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web-app
  template:
    metadata:
      labels:
        app: web-app
    spec:
      containers:
      - name: web-app
        image: your-docker-repo/web-app:latest
        ports:
        - containerPort: 8080
```

### Explanation of the Deployment File

- **apiVersion, kind, metadata:** Standard Kubernetes fields specifying the API version, resource type, and metadata (name and namespace).
- **replicas:** Specifies the number of pod replicas to run.
- **selector:** Defines which pods the deployment manages, matching those with the label `app=web-app`.
- **template:** Specifies the pod template, including metadata (labels) and specifications (containers).
- **containers:** Defines the container(s) to run in each pod. The `image` field specifies the Docker image to use, and `containerPort` specifies the port the application listens on within the container.

These configurations set up the necessary resources for handling HTTPS traffic and routing it to the appropriate pods in your Kubernetes cluster.

for an overall picture of understanding lets build a small project like a static HTML web page project through k8 including all above key components give step by step and explain the whole process and end give flow chart as well as

### Step-by-Step Guide to Deploy a Static HTML Web Page in Kubernetes

This guide will walk you through deploying a simple static HTML web page using Kubernetes, incorporating all the key components discussed: Ingress Controller, Service, and Deployment.

## Prerequisites

- Kubernetes cluster (e.g., Minikube, GKE, EKS, etc.)
- kubectl configured to interact with your cluster
- A static HTML file for the web page
- Basic understanding of Docker and Kubernetes

## Step 1: Create a Docker Image for the Static HTML Page

1. **Create an HTML file** (e.g., `index.html`):

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Static Web Page</title>
</head>
<body>
    <h1>Welcome to My Static Web Page</h1>
</body>
</html>
```

2. **Create a Dockerfile**:

```dockerfile
FROM nginx:alpine
COPY index.html /usr/share/nginx/html/index.html
```

3. **Build and push the Docker image**:

```sh
# Build the Docker image
docker build -t your-docker-repo/static-web:latest .

# Push the image to a Docker registry
docker push your-docker-repo/static-web:latest
```

## Step 2: Create Kubernetes Deployment

1. **Create a Deployment YAML file** (`deployment.yaml`):

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: static-web-deployment
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: static-web
  template:
    metadata:
      labels:
        app: static-web
    spec:
      containers:
      - name: static-web
        image: your-docker-repo/static-web:latest
        ports:
        - containerPort: 80
```

2. **Apply the Deployment**:

```sh
kubectl apply -f deployment.yaml
```

## Step 3: Create Kubernetes Service

1. **Create a Service YAML file** (`service.yaml`):

```yaml
apiVersion: v1
kind: Service
metadata:
  name: static-web-service
  namespace: default
spec:
  selector:
    app: static-web
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
  type: ClusterIP
```

2. **Apply the Service**:

```sh
kubectl apply -f service.yaml
```

## Step 4: Set Up Ingress Controller

1. **Install NGINX Ingress Controller**:

```sh
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/main/deploy/static/provider/cloud/deploy.yaml
```

2. **Create an Ingress Resource YAML file** (`ingress.yaml`):

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: static-web-ingress
  namespace: default
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  tls:
  - hosts:
    - your-domain.com
    secretName: tls-secret
  rules:
  - host: your-domain.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: static-web-service
            port:
              number: 80
```

3. **Apply the Ingress Resource**:

```sh
kubectl apply -f ingress.yaml
```

## Step 5: Configure TLS (Optional but Recommended)

1. **Create a TLS Secret** (assuming you have `tls.crt` and `tls.key` files):

```sh
kubectl create secret tls tls-secret --key tls.key --cert tls.crt --namespace default
```

## Step 6: Verify the Deployment

1. **Check the status of the pods**:

```sh
kubectl get pods -l app=static-web
```

2. **Check the status of the services**:

```sh
kubectl get svc
```

3. **Check the status of the ingress**:

```sh
kubectl get ingress
```

## Flowchart

```mermaid
flowchart TD
    A[Client Browser] -->|HTTPS Request| B[DNS Server]
    B -->|Resolve your-domain.com| C[Load Balancer (AWS ELB)]
    C -->|Forward HTTPS| D[NGINX Ingress Controller]
    D -->|TLS Termination & Routing| E[static-web-service]
    E -->|Forward to static-web-pods| F[static-web-pod-1]
    E -->|Forward to static-web-pods| G[static-web-pod-2]
    E -->|Forward to static-web-pods| H[static-web-pod-3]
    F -->|Process Request & Generate Response| E
    G -->|Process Request & Generate Response| E
    H -->|Process Request & Generate Response| E
    E --> D
    D --> C
    C --> A
```

## Explanation of the Process

1. **Client Browser**: A user opens a browser and navigates to `**https://your-domain.com**`.
2. **DNS Server**: Resolves the domain name to the IP address of the Load Balancer.
3. **Load Balancer (AWS ELB)**: Accepts incoming HTTPS requests and forwards them to the NGINX Ingress Controller.
4. **NGINX Ingress Controller**: Handles TLS termination, decrypting the HTTPS traffic, and routes the request based on defined Ingress rules.
5. **Kubernetes Service (static-web-service)**: Acts as an intermediary to forward the request to the appropriate pods, providing load balancing.
6. **Pods (static-web-pods)**: Each pod runs the static web application, processes the request, and generates a response. The response is then sent back through the same path to the client browser.

This setup ensures secure and efficient handling of web traffic, making it easy to manage and scale your static web application in Kubernetes.

give me flow chart of only this service or tolls

Certainly! Here's a simplified flowchart focusing on the key components and tools involved in deploying a static HTML web page project in Kubernetes:

```
mermaid
```

```
flowchart TD
    A[Client Browser] -->|HTTPS Request| B[DNS Server]
    B -->|Resolve your-domain.com| C[Load Balancer (AWS ELB)]
    C -->|Forward HTTPS| D[NGINX Ingress Controller]
    D -->|TLS Termination & Routing| E[static-web-service]
    E -->|Forward to static-web-pods| F[static-web-pod-1]
    E -->|Forward to static-web-pods| G[static-web-pod-2]
    E -->|Forward to static-web-pods| H[static-web-pod-3]
    F -->|Process Request & Generate Response| E
    G -->|Process Request & Generate Response| E
    H -->|Process Request & Generate Response| E
    E --> D
    D --> C
    C --> A
```

### Flowchart Explanation:

1. **Client Browser (A)**: Initiates an HTTPS request to `your-domain.com`.
2. **DNS Server (B)**: Resolves `your-domain.com` to the IP address of the Load Balancer (AWS ELB).
3. **Load Balancer (C)**: Accepts the incoming HTTPS request and forwards it to the NGINX Ingress Controller.
4. **NGINX Ingress Controller (D)**: Handles TLS termination, decrypts HTTPS traffic, and routes it based on defined Ingress rules.
5. **static-web-service (E)**: Kubernetes Service that directs traffic to one of the static web pods (`static-web-pod-1`, `static-web-pod-2`, `static-web-pod-3`).
6. **static-web-pods (F, G, H)**: Pods running the static web application. They process the request, generate a response, and send it back through the same path to the client browser.
7. **Paths**: Arrows represent the flow of HTTPS requests and responses between the components involved in serving the static web page.

This flowchart illustrates the sequence of actions from the client's request to the server's response, highlighting the roles of DNS resolution, load balancing, ingress routing, Kubernetes service handling, and pod execution in delivering the static HTML web page content securely and efficiently.