How to enable TLS for an ingress rule in Kubernetes?

Method 1: Self-signed certificate

Generate a self-signed certificate and private key:

```
openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 -keyout tls.key -out tls.crt -subj "/CN=arye.ir"
```

Create a secret containing the key and certificate:

```
kubectl create secret tls tls-secret --key tls.key --cert tls.crt
```

Method 2: Use Certbot

Use Certbot to generate a TLS certificate for your domain.

```
certbot --manual --preferred-challenges dns certonly -d arye.ir
```

Create a Kubernetes secret that contains the private key and the certificate

```
kubectl create secret tls tls-secret --key privkey.pem --cert cert.pem
```

Configure Ingress to Use the Certificate

Reference tls-secret secret in your Ingress resource

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-path
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
spec:
  tls:
  - hosts:
    - arye.ir
    secretName: tls-secret
  rules:
  - host: arye.ir
    http:
      paths:
        - path: /booklet
          pathType: Prefix
          backend:
            service:
              name: book-service
              port:
                name: http
...
```

This YAML manifest describes an Ingress resource that enables TLS for the host "arye.ir", redirects HTTP traffic to HTTPS, and defines a routing rule for the path "/booklet" to the backend service named "book-service" on the specified port

You can find more annotations in this link

https://github.com/kubernetes/ingress-nginx/blob/main/docs/user-guide/nginx-configuration/annotations.md

Method 3: Use Cert-manager

Cert-manager

Cert-manager is a certificate management controller for k8s. It helps with issuing and renewing certificates from various sources, such as Let's Encrypt, HashiCorp Vault, Venafi. cert-manager ensures certificates are valid and up to date, and will attempt to renew certificates at a configured time before expiry.

main features of cert-manager

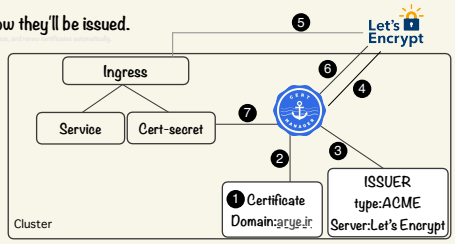
- Issuing and renewing certificates from a variety of sources
- DNS01 and HTTP01 ACME challenge solver support for Let's Encrypt certificates
- Issuing certificates for Certificate resources using CRDs
- Automated creation and updating of k8s Secrets with certificates
- Issuing certificates for Ingress resources with annotations

Cert-manager mainly uses two different custom Kubernetes resources (CRDs) to configure and control how it operates, as well as to store state. These resources are Issuers and Certificates.

Issuer is an object that represents a particular certificate authority or a specific method for issuing certificates. It defines the parameters and configurations required to request certificates. An Issuer can be used to issue certificates within a single namespace or cluster in Kubernetes. There are different types of issuers supported by CertManager, such as:

- ACME Issuer: This type of issuer integrates with the Automated Certificate Management Environment (ACME) protocol, which is commonly used by Let's Encrypt and other CAs. ACME issuers automate the process of obtaining and renewing certificates.
- CA Issuer: This type of issuer is used when you have an existing certificate authority (CA) that you want to use for issuing certificates. It requires you to provide the CA's certificate and private key.
- Self-Signed Issuer: This type of issuer is used when you want to generate self-signed certificates within the Kubernetes cluster. It is typically used for testing or development purposes

Certificates resources allow you to specify the details of the certificate you want to request. They reference an issuer to define how they'll be issued.



what happens when you create a Certificate resource in cert-manager:

1. You create a Certificate resource with details like the domain name, secret name to store certificate, and reference to the Issuer
2. The cert-manager controller sees the new Certificate and kicks off the issuance process
3. cert-manager first checks if the referenced Issuer exists and is valid. The Issuer has the details for the certificate authority
4. cert-manager requests the certificate authority (CA) like Let's Encrypt to issue a certificate for the requested domain
5. The CA validates that you own/control the domain name by performing a challenge. For example, with HTTP challenge, you need to have a temporary file served on the domain
6. Once domain ownership is validated, the CA issues the signed certificate. The certificate is returned to cert-manager
7. cert-manager takes the certificate and creates or updates the Kubernetes secret defined in the Certificate. This secret will contain the certificate and private key.

How can I issue a certificate for the domain arye.ir using cert-manager from Let's Encrypt?

Configure Let's Encrypt Issuer

Cert-manager uses 'Issuer' or 'ClusterIssuer' resources to represent certificate authorities. We'll create a 'ClusterIssuer' for Let's Encrypt.

```yaml
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
spec:
  acme:
    # The ACME server URL
    server: https://acme-v02.api.letsencrypt.org/directory
    # Email address used for ACME registration
    email: your-email@your-domain.com
    # Name of a secret used to store the ACME account private key
    privateKeySecretRef:
      name: letsencrypt-prod
    # Enable the HTTP-01 challenge provider
    solvers:
    - http01:
        ingress:
          class: nginx
```

privateKeySecretRef specifies the name of the Kubernetes secret that will be used to store the private key for the certificate.

solvers specifies the method for verifying ownership of the domain. In this case, we are using the HTTP-01 challenge, which involves creating a temporary file in the web root of the domain and responding to an HTTP request to that file. The ingress field specifies that we will use an Ingress resource to serve the challenge.

this ClusterIssuer can be referenced by other resources like Certificate or ingress to automatically generate and manage self-signed certificates.

```yaml
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: selfsigned-issuer
spec:
  selfSigned: {}
```

Issue a Certificate

Create a certificate resource to obtain the certificate from Let's Encrypt for the specified domain.

```yaml
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: arye-ir-cert
  namespace: default
spec:
  secretName: arye-ir-tls
  issuerRef:
    name: letsencrypt-prod
    kind: ClusterIssuer
  commonName: arye.ir
  dnsNames:
  - arye.ir
  - *.arye.ir
  duration: 90d
```

After a few moments, cert-manager should issue a certificate for your domain and store it in the secret specified in the certificate resource. You can verify that the certificate has been issued by checking the contents of the secret:

```
kubectl describe secret arye-ir-tls
```

```
Data
====
tls.crt:  2316 bytes
tls.key:  1704 bytes
```

secretName specifies the name of the Kubernetes secret that will be used to store the TLS certificate and private key.

issuerRef specifies the name and kind of the Kubernetes resource that is used as the issuer for this certificate. In this case, we are using the previously defined letsencrypt-prod ClusterIssuer

commonName specifies the common name for the TLS certificate. In this case, it is set to arye.ir.

dnsNames specifies the list of DNS names for which the TLS certificate should be issued. In this case, we are issuing the certificate for arye.ir and all subdomains of the specified domain

Configure Ingress to Use the Certificate

Your Ingress configuration should use the secret arye-ir-tls for its TLS configuration

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: arye.ir-ingress
spec:
  tls:
  - hosts:
    - arye.ir
    - *.arye.ir
    secretName: arye-ir-tls
  rules:
  - host: arye.ir
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: your-service-name
            port:
              number: 80
```

Validate the Setup

Check if the certificate has been issued successfully

```
kubectl describe certificate arye-ir-cert
```

```
Name:         arye-ir-cert
Namespace:    default
Labels:       <none>
Annotations:  <none>
API Version:  cert-manager.io/v1
Kind:         Certificate
...
Spec:
  Common Name:  arye.ir
  DNS Names:
    arye.ir
  Issuer Ref:
    Group:      cert-manager.io
    Kind:       ClusterIssuer
    Name:       letsencrypt-prod
  Secret Name:  tls-secret
Status:
  Conditions:
    Last Transition Time:  2023-09-27T10:01:00Z
    Message:               Certificate is up to date and has not expired
...
  Not After:               2024-09-27T10:00:00Z
Events:                    <none>
```

You can configure TLS for Ingress using annotations instead of Certificate resources

- Use annotations for basic, single TLS certificate configuration. Simpler, but less flexible.
- Use Certificate resources for multiple certificates, automation, and advanced management. More complex, but more flexible and powerful

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: arye.ir-ingress
annotations:
    cert-manager.io/cluster-issuer: "letsencrypt-prod"
spec:
  tls:
  - hosts:
    - arye.ir
    - *.arye.ir
    secretName: arye-ir-tls
  rules:
  - host: arye.ir
...
```

cert-manager.io/cluster-issuer: References the Issuer resource in cert-manager that will be used to obtain the certificate

Add-ons

Kubernetes has a rich ecosystem of add-ons and extensions that provide additional functionality and features to enhance and extend the capabilities of a Kubernetes cluster. So far, we have covered a few of these add-ons in the booklet. Now, let's introduce some additional add-ons that can further enhance your Kubernetes experience:

Argo CD is a powerful open-source tool designed for Kubernetes, enabling GitOps continuous delivery. It simplifies application deployment and management by utilizing a declarative approach. With Argo CD, you can define the desired state of your applications using Kubernetes manifests stored in a Git repository. It provides a user-friendly graphical interface to monitor application status, track changes, and roll back if needed. By following GitOps principles, Argo CD ensures that your cluster's configuration matches the desired state defined in the repository, automatically deploying and updating applications.

Service mesh add-ons, like Istio and Linkerd, are powerful tools that enhance the networking and observability capabilities of microservices within a Kubernetes cluster. By integrating transparently with the cluster, they offer advanced features for traffic management, security, and distributed tracing. These service mesh solutions enable fine-grained control over traffic routing, load balancing, and fault tolerance mechanisms, ensuring efficient and reliable communication between microservices. With built-in security features like mutual TLS authentication and encryption, they provide robust protection for service-to-service communication. Additionally, service mesh add-ons enable comprehensive observability with distributed tracing, metrics collection, and logging, allowing for deep insights into the behavior and performance of microservices.

Rook and Longhorn are two notable storage-related add-ons for Kubernetes. Rook is a cloud-native storage orchestrator that enables the deployment and management of various storage solutions as native Kubernetes resources. It automates the provisioning, scaling, and lifecycle management of distributed storage systems like Ceph, CockroachDB, and more. On the other hand, Longhorn is a lightweight, open-source distributed block storage system built for Kubernetes. It provides reliable, replicated block storage for stateful applications, offering features like snapshots, backups, and volume expansion. Together, Rook and Longhorn empower Kubernetes users to easily deploy and manage resilient, scalable, and persistent storage solutions within their clusters, enhancing the availability and data management capabilities of their applications.

Monitoring and logging add-ons, such as Prometheus, facilitate the collection and storage of time-series data and metrics from diverse Kubernetes components and applications, enabling comprehensive analysis and alerting capabilities. Additionally, Fluentd serves as a dependable log aggregation tool, simplifying the gathering, parsing, and forwarding of logs from multiple sources to ensure centralized and scalable log management. The ELK (Elasticsearch, Logstash, and Kibana) stack offers a comprehensive solution for monitoring and logging, utilizing Elasticsearch for efficient log indexing and searching, Logstash for data processing and filtering, and Kibana for visualizing and analyzing log data. Together, these add-ons provide Kubernetes users with powerful tools for monitoring performance, detecting issues, and gaining valuable insights to optimize their Kubernetes environments.

Additionally, The CNCF landscape is an excellent resource for discovering and exploring a vast array of add-ons and tools within the cloud-native ecosystem. It offers a visual representation of different projects and categories, allowing users to navigate through various technologies that can enhance their Kubernetes deployments. Whether you're looking for monitoring and observability tools, networking solutions, or storage options, the CNCF landscape provides a comprehensive overview of the available options. By exploring the CNCF landscape, you can expand your knowledge and make informed decisions about incorporating the right add-ons and tools into your Kubernetes and cloud-native environments. It's a valuable resource for staying up-to-date with the latest innovations and finding the best solutions to meet your specific needs.