Deploying Traefik as Ingress Controller for Your Kubernetes Cluster

Kirill Goltsman

You might already know from our previous tutorials about how to use Kubernetes Services to distribute traffic between multiple backends. In the production environment, however, we might also need to control external traffic going to our cluster from the Internet. This is precisely what Ingress does.

The main purpose of Ingress is to expose HTTP and HTTPS routes from outside the cluster to services running in that cluster. This is the same as to say that Ingress controls how the external traffic is routed to the cluster. Let's give a practical example to illustrate the concept of Ingress more clearly.

Let's imagine that you have several microservices (small applications communicating with each other) in your Kubernetes cluster. These services can be accessed from within the cluster, but we might also want our users to access them from outside the cluster as well. What we therefore need to do is to associate each HTTP(S) route (e.g., `service.yourdomain.com`) with the corresponding backend using the reverse proxy and load balance between different instances (e.g., Pods) of this service. At the same time, given the ever-changing nature of Kubernetes, we would want to track changes in service backends to be able to re-associate those HTTP routes to new Pod instances when new Pods are added or removed.

Using the Ingress resource and the associated Ingress Controller you could achieve the following:

- Point your domain `app.domain.com` to the microservice `app` in your private network.
- Point the path `domain.com/web` to the microservice `web` in your private network.
- Point your domain `backend.domain.com` to the microservice `backend` in your private network and load balance between multiple instances (Pods) of that microservice.

Now you understand the importance of Ingress. It's helpful for pointing HTTP routes to specific microservices inside the Kubernetes cluster.

Traffic routing, however, is not the sole purpose of Ingress in Kubernetes. For example, Ingress can be also configured to load balance traffic to your apps, terminate SSL, perform name-based virtual hosting, split traffic between different services, set service access rules, etc.

Kubernetes has a special Ingress API resource that supports all this functionality. However, simply creating an Ingress API resource will have no effect. You will also need an Ingress controller to implement Ingress in Kubernetes. Kubernetes currently supports a number of Ingress controllers including Contour, HAProxy based ingress controller jcmoraisjr/haproxy-ingress , NGINX Ingress Controller for Kubernetes, and Traefik among others (see the official Kubernetes documentation to learn more).

In this article, we'll walk you through creating Ingress using the Traefik Ingress Controller. It acts as a modern HTTP reverse proxy and a load balancer that simplifies deployment of microservices. Traefik has great support for such systems and environments as Docker, Marathon, Consul, Kubernetes, Amazon ECS, etc.

Traefik is especially useful for such flexible systems as Kubernetes where services are added, removed, or upgraded many times a day. It listens to the service registry/orchestrator API and instantly generates/updates the routes, so your microservices are always connected to the outside world without manual configuration.

In addition to that, Traefik supports multiple load balancing algorithms, HTTPS via Let's Encrypt (wildcard certificates support), circuit breakers, WebSockets, GRPC, and multiple metrics providers (Rest, Prometheus, Statsd, Datadog, InfluxDB, etc.). For more information about the available feature in Traefik, see its official documentation.

Ingress Resource

Before starting the tutorial, let's briefly discuss how Ingress resource works. Below is the minimum Ingress example that implicitly uses the Nginx Ingress Controller.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-example
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /microservice1
        backend:
          serviceName: test
          servicePort: 80
```

The Ingress manifest above contains a set of HTTP rules that tell the controller how to route the traffic.

Each rule contains the following info:

- **An optional host**. If the host is not specified (like in the example above), the rule applies to all inbound HTTP traffic through the IP address specified. If a host is provided (for example, `yourhost.com`), the rule applies only to that host.
- **A list of paths** (for example, `/microservice1`) that point to the associated backends defined with a `serviceName` and `servicePort` . Obviously, a Service that connects multiple Pods together should be created for the rule to work.
- **A backend**. HTTP (and HTTPS) requests to the Ingress matching the host and path of a given rule will be routed to the backend Service specified in that rule.

In the example above, we've configured a backend named "test" that will receive all the traffic from the `/microservice1` path. However, we can also configure a default backend that will service any user requests that do not match a path in the spec. Also, if no rules are defined, an Ingress will route all traffic to a single default backend. For example:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
spec:
  backend:
    serviceName: defaultbackend
    servicePort: 80
```

In this example, all the traffic will be forwarded to the default backend `defaultbackend`. Now, as we understand the basics of Ingress resource definition, let's move on to some concrete examples.

Tutorial

As we mentioned earlier, defining an Ingress resource won't take effect unless you implement the Ingress controller. In this tutorial, we will set up Traefik as the Ingress controller in your Kubernetes cluster.

To complete examples used below, you'll need the following prerequisites:

- A running Kubernetes cluster. See Supergiant documentation for more information about deploying a Kubernetes cluster with Supergiant. As an alternative, you can install a single-node Kubernetes cluster on a local system using Minikube.
- A **kubectl** command line tool installed and configured to communicate with the cluster. See how to install **kubectl** here.

Note: all examples below assume that you run Minikube as your Kubernetes cluster on the local machine.

Step 1: Enabling RBAC

We first need to grant some permissions to Traefik to access Pods, Endpoints, and Services running in your cluster. To this end, we will be using a `ClusterRole` and a `ClusterRoleBinding` resources. However, you can also use the least-privileges approach with the namespace-scoped `RoleBindings`. In general, this is a preferred approach if a cluster's namespaces do not change dynamically and if Traefik is not supposed to watch all cluster namespaces.

Let's first create a new `ServiceAccount` to provide Traefik with the identity in your cluster.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: traefik-ingress
  namespace: kube-system
```

To create the ServiceAccount, save the manifest above in the `traefik-service-acc.yaml` and run:

```
kubectl create -f traefik-service-acc.yamlserviceaccount "traefik-ingress" created
```

Next, let's create a `ClusterRole` with a set of permissions which will be applied to the Traefik `ServiceAccount`. The `ClusterRole` will allow Traefik to manage and watch such resources as Services, Endpoints, Secrets, and Ingresses across all namespaces in your cluster.

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: traefik-ingress
rules:
  - apiGroups:
      - ""
    resources:
      - services
      - endpoints
      - secrets
    verbs:
      - get
      - list
      - watch
  - apiGroups:
      - extensions
    resources:
      - ingresses
    verbs:
      - get
      - list
      - watch
```

Save this spec to the file `traefik-cr.yaml` and run:

```
kubectl create -f traefik-cr.yamlclusterrole.rbac.authorization.k8s.io “traefik-ingress” created
```

Finally, to enable these permissions, we should bind the `ClusterRole` to the Traefik `ServiceAccount`. This can be done using the `ClusterRoleBinding` manifest:

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: traefik-ingress
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: traefik-ingress
subjects:
- kind: ServiceAccount
  name: traefik-ingress
  namespace: kube-system
```

Save this spec to the `traefik-crb.yaml` and run the following command:

```
kubectl create -f traefik-crb.yamlclusterrolebinding.rbac.authorization.k8s.io “traefik-ingress” created
```

Step 2: Deploy Traefik to a Cluster

Next, we will deploy Traefik to the Kubernetes cluster. The official Traefik documentation for Kubernetes supports three types of deployments: using a Deployment object, using a DaemonSet object, or using the Helm chart.

In this tutorial, we’ll be using the Deployment manifest. Deployments have a number of advantages over other options. For example, they secure better scalability (up and down scaling) and come with good support for rolling updates.

Let’s take a look at the Deployment manifest:

```
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: traefik-ingress
  namespace: kube-system
  labels:
    k8s-app: traefik-ingress-lb
spec:
  replicas: 1
  selector:
    matchLabels:
      k8s-app: traefik-ingress-lb
  template:
    metadata:
      labels:
        k8s-app: traefik-ingress-lb
        name: traefik-ingress-lb
    spec:
      serviceAccountName: traefik-ingress
      terminationGracePeriodSeconds: 60
      containers:
      - image: traefik
        name: traefik-ingress-lb
        ports:
        - name: http
          containerPort: 80
        - name: admin
          containerPort: 8080
        args:
        - --api
        - --kubernetes
        - --logLevel=INFO
```

This Deployment will create one Traefik replica in the `kube-system` namespace. The Traefik container will be using ports 80 and 8080 specified in this manifest.

Save this manifest to the `traefik-deployment.yaml` file and create the Deployment running the following command:

```
kubectl create -f traefik-deployment.yaml
deployment.extensions “traefik-ingress” created
```

Now, let’s check to see if the Traefik Pods were successfully created:

```
kubectl --namespace=kube-system get podsNAME                          READY     STATUS    RESTARTS   AGE....storage-provisioner           1/1       Running   3          23dtraefik-ingress-54d6d8d9cc-ls6cs 1/1       Running   0          1m
```

As you see, the Deployment Controller launched one Traefik replica, and it is currently running. Awesome!

Step 3: Create NodePorts for External Access

Let’s create a Service to access Traefik from outside of the cluster. To this end, we need a Service that exposes two `NodePorts`.

```
kind: Service
apiVersion: v1
metadata:
  name: traefik-ingress-service
  namespace: kube-system
spec:
  selector:
    k8s-app: traefik-ingress-lb
  ports:
    - protocol: TCP
      port: 80
      name: web
    - protocol: TCP
      port: 8080
      name: admin
  type: NodePort
```

Save this manifest to `traefik-svc.yaml` and create the Service:

```
kubectl create -f traefik-svc.yaml
service “traefik-ingress-service” created
```

Now, let’s verify that the Service was created:

```
kubectl describe svc traefik-ingress-service --namespace=kube-system
Name:                     traefik-ingress-service
Namespace:                kube-system
Labels:                   <none>
Annotations:              <none>
Selector:                 k8s-app=traefik-ingress-lb
Type:                     NodePort
IP:                       10.102.215.64
Port:                     web  80/TCP
TargetPort:               80/TCP
NodePort:                 web  30565/TCP
Endpoints:                172.17.0.6:80
Port:                     admin  8080/TCP
TargetPort:               8080/TCP
NodePort:                 admin  30729/TCP
Endpoints:                172.17.0.6:8080
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

As you see, now we have two NodePorts (“web” and “admin”) that route to the 80 and 8080 container ports of the Traefik Ingress controller. The “**admin**” NodePort will be used to access the Traefik Web UI and the “**web**” NodePort will be used to access services exposed via Ingress.

Step 4: Accessing Traefik

To access the Traefik Web UI in the browser, you can use the “admin” NodePort 30729 (please note that your NodePort value might differ). Because we haven’t added any frontends, the UI will be empty at the moment.

We’ll also get a 404 response for Ingress endpoint because we haven’t yet given Traefik any configuration.

```
curl $(minikube ip):30565404 page not found
```

Step 5: Adding Ingress to the Cluster

Now we have Traefik as the Ingress Controller in the Kubernetes cluster. However, we still need to define the Ingress resource and a Service that exposes Traefik Web UI.

Let’s first create a Service:

```
apiVersion: v1
kind: Service
metadata:
  name: traefik-web-ui
  namespace: kube-system
spec:
  selector:
    k8s-app: traefik-ingress-lb
  ports:
  - name: web
    port: 80
    targetPort: 8080
```

Save this manifest to `traefik-webui-svc.yaml` and run:

```
kubectl create -f traefik-webui-svc.yaml
service “traefik-web-ui” created
```

Let’s verify that the Service was created:

```
kubectl describe svc traefik-web-ui --namespace=kube-system
Name:              traefik-web-ui
Namespace:         kube-system
Labels:            <none>
Annotations:       <none>
Selector:          k8s-app=traefik-ingress-lb
Type:              ClusterIP
IP:                10.98.230.58
Port:              web  80/TCP
TargetPort:        8080/TCP
Endpoints:         172.17.0.6:8080
Session Affinity:  None
Events:            <none>
```

As you see, the Service was given a `ClusterIP` of `10.98.230.58` and was assigned ports specified in the manifest.

Next, we need to create an Ingress resource pointing to the Traefik Web UI backend.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: traefik-web-ui
  namespace: kube-system
spec:
  rules:
  - host: traefik-ui.minikube
    http:
      paths:
      - path: /
        backend:
          serviceName: traefik-web-ui
          servicePort: web
```

In essence, this Ingress routes all requests to `traefik-ui.minikube` host to the Traefik Web UI exposed by the Service created in the step above.

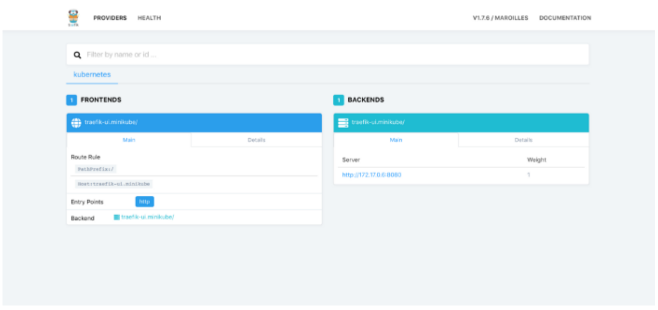
Save the spec to the `traefik-ingress.yaml` and run:

```
kubectl create -f traefik-ingress.yaml
ingress.extensions “traefik-web-ui” created
```

To make the Traefik Web UI accessible in the browser via the `traefik-ui.minikube` , we need to add a new entry to our `/etc/hosts` file. The entry will contain the Minikube IP and the name of the host. You can get the IP address of your minikube instance by running `minikube ip` and then save the name of a new host to `/etc/hosts` file like this:

```
echo "$(minikube ip) traefik-ui.minikube" | sudo tee -a /etc/hosts192.168.99.100 traefik-ui.minikube
```

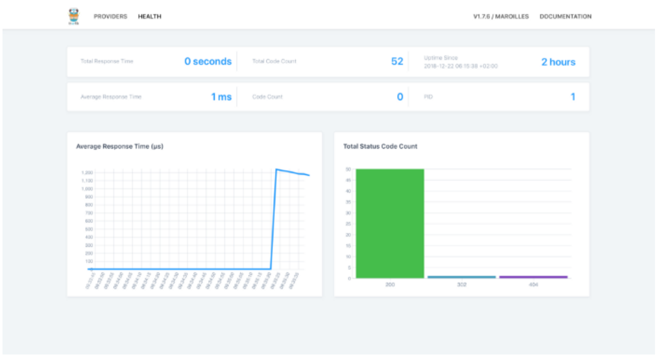
You should now be able to visit http://traefik-ui.minikube:<AdminNodePort> in the browser and view the Traefik web UI. Don’t forget to append the “admin” `NodePort` to the host address.



Route Rule
PathPrefix:/
Host:traefik-ui.minikube
Entry Points http
Backend traefik-ui.minikube/

Server Weight
http://172.17.0.6:8080 1

Inside the dashboard, you can click on the Health link to view the health of the application:



Total Response Time 0 seconds
Total Code Count 52
Uptime Since 2018-12-22 06:15:38 +02:00 2 hours
Average Response Time 1 ms
Code Count 0
PID 1

Step 6: Implementing Name-Based Routing

Now, let’s demonstrate how Traefik Ingress Controller can be used to set up name-based routing for a list of frontends. We will create three Deployments with simple single-page websites displaying images of animals: bear, hare, and moose.

```
---
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: bear
  labels:
    app: animals
    animal: bear
spec:
  replicas: 2
  selector:
    matchLabels:
      app: animals
      task: bear
  template:
    metadata:
      labels:
        app: animals
        task: bear
        version: v0.0.1
    spec:
      containers:
      - name: bear
        image: supergiantkir/animals:bear
        ports:
        - containerPort: 80
---
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: moose
  labels:
    app: animals
    animal: moose
spec:
  replicas: 2
  selector:
    matchLabels:
      app: animals
      task: moose
  template:
    metadata:
      labels:
        app: animals
        task: moose
        version: v0.0.1
    spec:
      containers:
      - name: moose
        image: supergiantkir/animals:moose
        ports:
        - containerPort: 80
---
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: hare
  labels:
    app: animals
    animal: hare
spec:
  replicas: 2
  selector:
    matchLabels:
      app: animals
      task: hare
  template:
    metadata:
      labels:
        app: animals
        task: hare
        version: v0.0.1
    spec:
      containers:
      - name: hare
        image: supergiantkir/animals:hare
        ports:
        - containerPort: 80
```

Each Deployment will have two Pod replicas, and each Pod will serve the “animal” websites on the `containerPort` 80.

Let’s save these Deployment manifests to `animals-deployment.yaml` and run:

```
kubectl create -f animals-deployment.yamldeployment.extensions “bear” created
deployment.extensions “moose” created
deployment.extensions “hare” created
```

Now, let’s create a Service for each Deployment to make the Pods accessible:

```
---
apiVersion: v1
kind: Service
metadata:
  name: bear
spec:
  ports:
  - name: http
    targetPort: 80
    port: 80
  selector:
    app: animals
    task: bear
---
apiVersion: v1
kind: Service
metadata:
  name: moose
spec:
  ports:
  - name: http
    targetPort: 80
    port: 80
  selector:
    app: animals
    task: moose
---
apiVersion: v1
kind: Service
metadata:
  name: hare
  annotations:
    traefik.backend.circuitbreaker: "NetworkErrorRatio() > 0.5"
spec:
  ports:
  - name: http
    targetPort: 80
    port: 80
  selector:
    app: animals
    task: hare
```

Note: the third Service uses the circuit breaker annotation. A circuit breaker is the Traefik feature that prevents high loads on the failing server. In this example, we prevent the high loads on servers with the greater than 50%. When this condition matches, CB enters a Tripped state where it responds with predefined HTTP status code or redirects to another frontend.

Save these Service manifests to `animals-svc.yaml` and run:

```
kubectl create -f animals-svc.yamlservice “bear” created
service “moose” created
service “hare” created
```

Finally, let’s create an Ingress with three frontend-backend pairs for each Deployment. `bear.minikube` , `moose.minikube` , and `hare.minikube` will be our frontends pointing to corresponding backend Services.

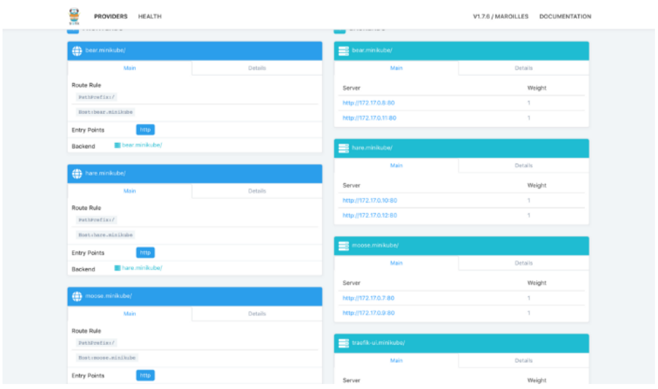
The Ingress manifests looks as follows:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: animals
  annotations:
    kubernetes.io/ingress.class: traefik
spec:
  rules:
  - host: hare.minikube
    http:
      paths:
      - path: /
        backend:
          serviceName: hare
          servicePort: http
  - host: bear.minikube
    http:
      paths:
      - path: /
        backend:
          serviceName: bear
          servicePort: http
  - host: moose.minikube
    http:
      paths:
      - path: /
        backend:
          serviceName: moose
          servicePort: http
```

Save this spec to `animals-ingress.yaml` and run:

```
kubectl create -f animals-ingress.yaml
ingress.extensions “animals” created
```

Now, inside the Traefik dashboard and you should see a frontend for each host along with a list of corresponding backends.



If you edit your `/etc/hosts` again you should be able to access the animal websites in your browser.

```
echo “$(minikube ip) bear.minikube hare.minikube moose.minikube” | sudo tee -a /etc/hosts
```

You should use the “web” NodePort to access specific sites. For example, `http://bear.minikube:<WebNodePort>`.

We can also reconfigure three frontends to serve under one domain like this:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: all-animals
  annotations:
    kubernetes.io/ingress.class: traefik
    traefik.frontend.rule.type: PathPrefixStrip
spec:
  rules:
  - host: animals.minikube
    http:
      paths:
      - path: /bear
        backend:
          serviceName: bear
          servicePort: http
      - path: /moose
        backend:
          serviceName: moose
          servicePort: http
      - path: /hare
        backend:
          serviceName: hare
          servicePort: http
```

If you activate this Ingress, all three animals will be accessible under one domain — `animals.minikube` — using corresponding paths. Don’t forget to add this domain to `/etc/hosts` .

```
echo “$(minikube ip) animals.minikube” | sudo tee -a /etc/hosts
```

Note: We are configuring Traefik to strip the prefix from the URL path with the `traefik.frontend.rule.type` annotation. This way we can use the containers from the previous example without modification. Because of the `traefik.frontend.rule.type: PathPrefixStrip rule` you have to use: `http://animals.minikube:32484/moose/` instead of `http://animals.minikube:32484/moose` (add forward slash to the path).

Step 7: Implementing Traffic Splitting

With Traefik, users can split Ingress traffic in a controlled manner between multiple deployments using *service weights*. This feature can be used for canary releases that should initially receive a small but ever-increasing fraction of traffic over time.

Let’s split the Traefik between two microservices using the following manifest:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    traefik.ingress.kubernetes.io/service-weights: |
      animals-app: 99%
      animals-app-canary: 1%
  name: animals-app
spec:
  rules:
  - http:
      paths:
      - backend:
          serviceName: animals-app
          servicePort: 80
        path: /
      - backend:
          serviceName: animals-app-canary
          servicePort: 80
        path: /
```

Take note of the `traefik.ingress.kubernetes.io/service-weights` annotation. It specifies how the traffic should be split between the referenced backend services, `animals-app` and`animals-app-canary` . Traefik will route 99% of user requests to the Pods backed by the `animals-app` deployment, and it will route 1% to Pods backed by the `animals-app-canary` deployment.

There are a few conditions to be met for this setup to work correctly:

- All service backends must share the same path and host.
- The total percentage of requests shared across service backends should add up to 100%.
- The percentage values are interpreted as floating point numbers to a supported precision. Currently, Traefik supports 3 decimal places for the weights.

Conclusion

As you’ve seen, Ingress is a powerful tool for routing external traffic to corresponding backend services in your Kubernetes cluster. Users can implement Ingress using a number of Ingress controllers supported by Kubernetes. In this tutorial, we focused on Traefik Ingress controller that supports name-based routing, load balancing, and other common tasks of Ingress controllers. Stay tuned to our blog to learn more Traefik features in Kubernetes.