Kubernetes rolling updates, rollbacks and multi-environments

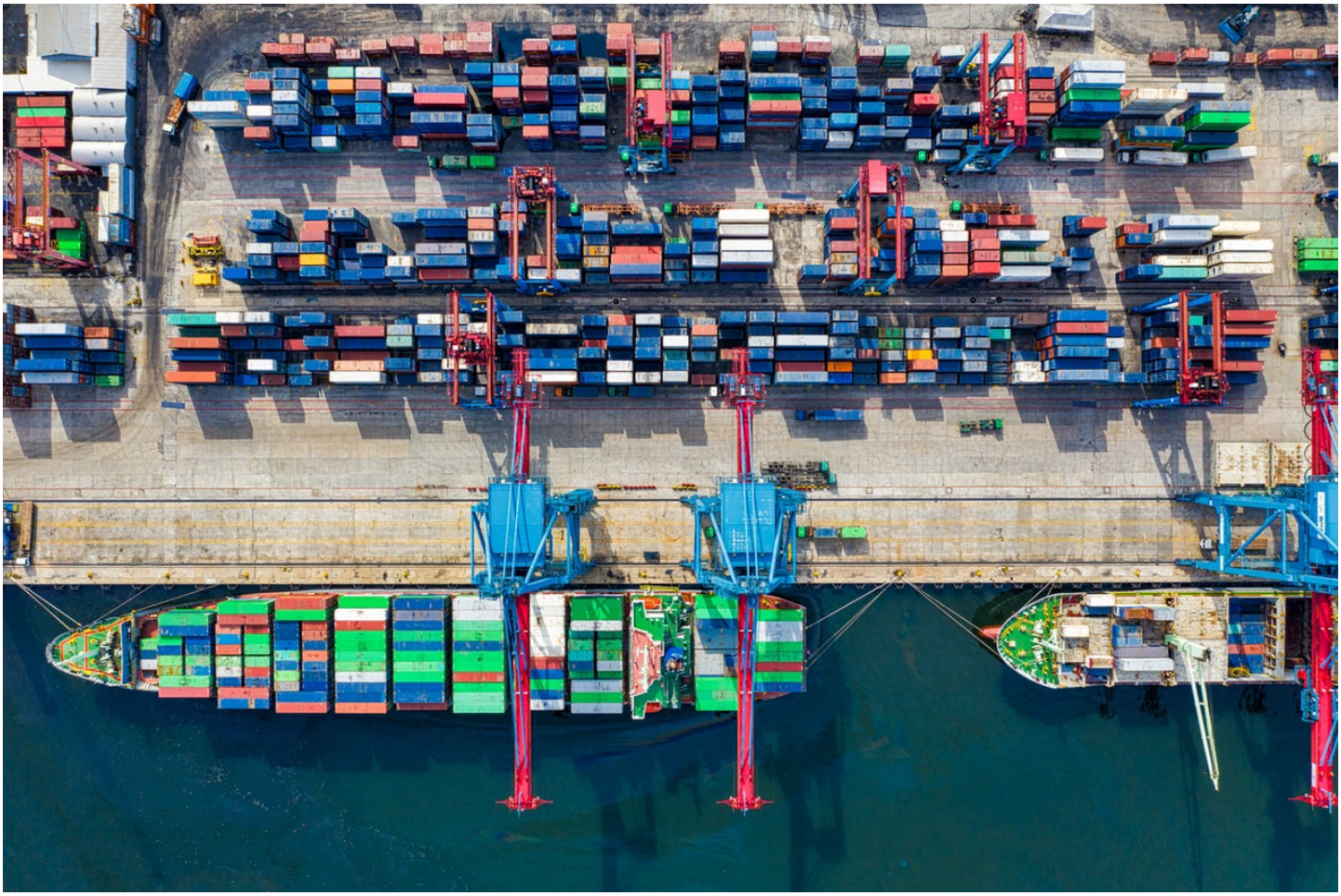


Photo by Tom Fisk from Pexels

On previous post (https://itnext.io/deploy-an-app-on-kubernetes-gke-with-kong-ingress-letsencrypt-and-cloudflare-94913e127c2b) we learned how to deploy an application with 2 micro-services (frontend and backend) to Kubernetes, using Kong Ingress, LetsEncrypt to provide TLS certificates and Cloudflare for proxy and extra security.

In this post we want to do some updates to our deployed application, roll them back in the case of errors and last but not least use multiple environments so we can test our application before deploying to production.

First I will re-deploy my original application. (I always delete un-used applications, no need to spend money on hosting them)

One nifty feature of `kubectl` is you can concatenate all resource files and apply them on bulk. So this is my file:

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: outsrc
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: outsrc-dev-ingress
  namespace: outsrc
  annotations:
    kubernetes.io/ingress.class: kong
    kubernetes.io/tls-acme: 'true'
    cert-manager.io/cluster-issuer: letsencrypt-production
spec:
  tls:
    - secretName: outsrc-dev-tls
      hosts:
        - outsrc.dev
  rules:
    - host: outsrc.dev
      http:
        paths:
          - path: /api
            backend:
              serviceName: service-backend
              servicePort: 3000
          - path: /
            backend:
              serviceName: service-frontend
              servicePort: 3000
---
apiVersion: v1
kind: Service
metadata:
  name: service-frontend
  namespace: outsrc
  labels:
    service: front
spec:
```

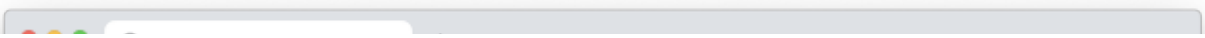
```yaml
  selector:
    service: front
  ports:
    - port: 3000
      protocol: TCP
      targetPort: 3000
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-frontend
  namespace: outsrc
  labels:
    service: front
spec:
  replicas: 2
  selector:
    matchLabels:
      service: front
  template:
    metadata:
      labels:
        service: front
    spec:
      containers:
        - name: frontend-container
          image: 'gcr.io/outsrc/outsrc-demo-front:1.0.0'
          imagePullPolicy: Always
          ports:
            - containerPort: 3000
          env:
            - name: API_URL
              value: 'https://outsrc.dev/api'
            - name: PORT
              value: '3000'
---
apiVersion: v1
kind: Service
metadata:
  name: service-backend
  namespace: outsrc
  labels:
    service: back
spec:
  selector:
    service: back
  ports:
    port: 3000
```

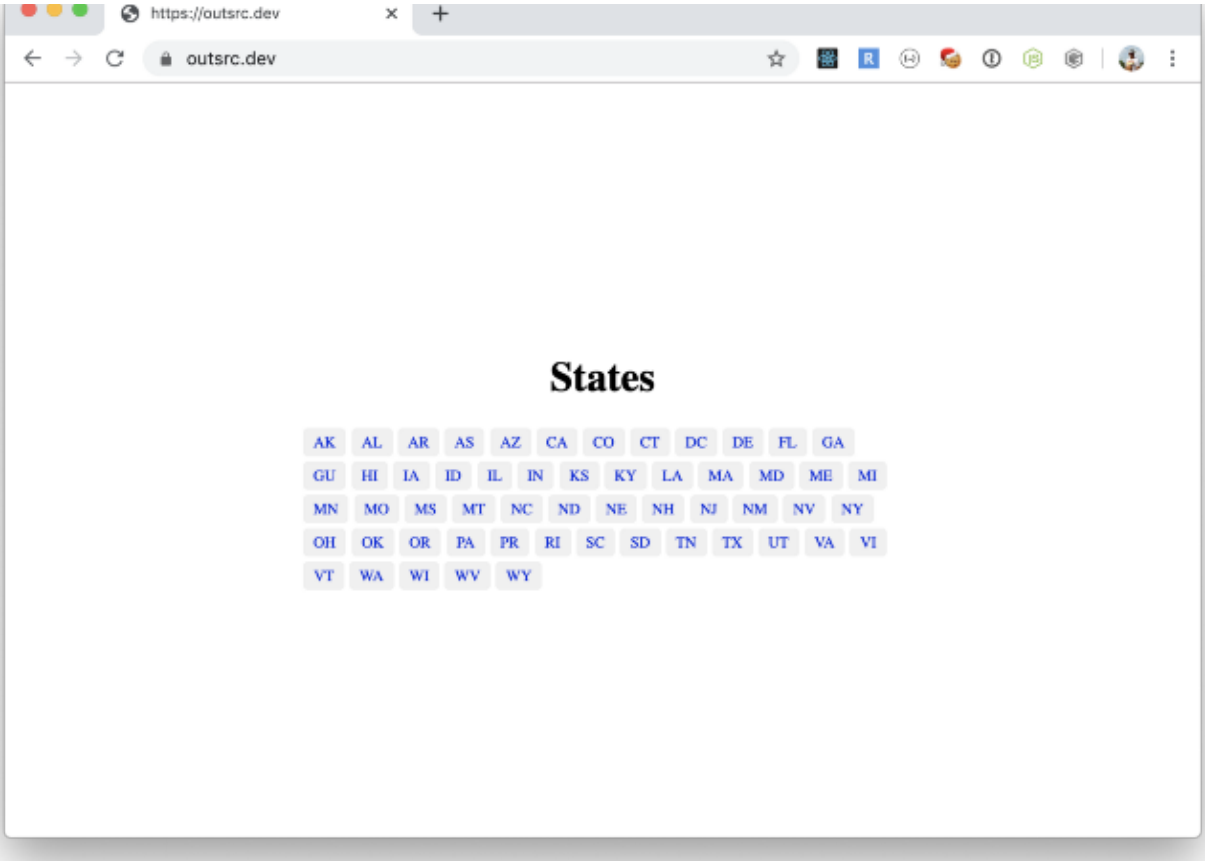
```
    - port: 3000
      protocol: TCP
      targetPort: 3000
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-backend
  namespace: outsrc
  labels:
    service: back
spec:
  replicas: 2
  selector:
    matchLabels:
      service: back
  template:
    metadata:
      labels:
        service: back
    spec:
      containers:
        - name: backend-container
          image: 'gcr.io/outsrc/outsrc-demo-back:1.0.0'
          imagePullPolicy: Always
          ports:
            - containerPort: 3000
          env:
            - name: PORT
              value: '3000'
```

to apply it:

```
$ kubectl apply -f outsrc.yml
namespace/outsrc created
ingress.extensions/outsrc-dev-ingress created
service/service-frontend created
deployment.apps/deployment-frontend created
service/service-backend created
deployment.apps/deployment-backend created
```

And after a couple seconds (container running, TLS certs emitted) we have the application back online:





outsrc.dev deployed to Kubernetes

So far all good. Now, we need to make some changes to the Application. We will add a map page containing the US map, we will link the map from both the main page and the state page. After adding the code for this feature and dockerize it:

```
$ docker build -t outsrc-demo-front .
...

$ docker tag outsrc-demo-front:latest gcr.io/outsrc/outsrc-demo-
front:1.1.0

$ docker push gcr.io/outsrc/outsrc-demo-front:1.1.0
...
```

Notice the version is different, We will use docker image tags to do a rolling update.

Rolling updates

Every time we want to update the application we deployed on our kubernetes cluster we change our deployment resource files and update them. Each time a change is detected a rolling update will be performed.

To avoid downtime kubernetes will update each replica of our running container one by one and re-routing the services on top.

To check updates history:

```
$ kubectl rollout history deployment/deployment-frontend
deployment.extensions/deployment-frontend
REVISION  CHANGE-CAUSE
1         <none>
```

Lets update the frontend container image tag:

```
$ kubectl set image deployment/deployment-frontend frontend-
container=gcr.io/outsrc/outsrc-demo-front:1.1.0
deployment.extensions/deployment-frontend image updated
```

We could also modify the deployment resource file, change the image tag and apply it via `kubectl` . This is my preferred way to handle updates, since it keeps the source of truth on the resource descriptor files.

Watch the update:

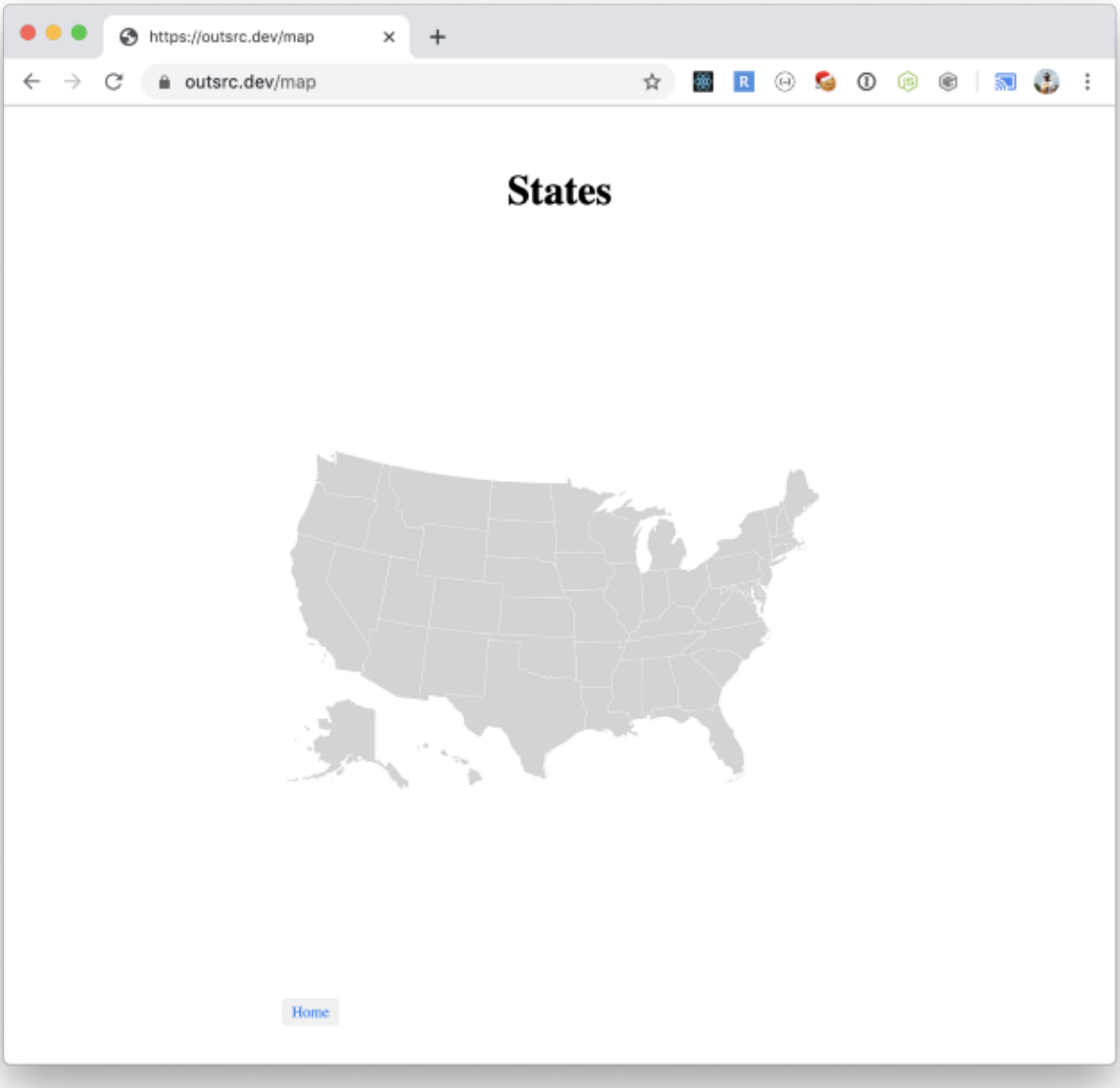
```
$ kubectl rollout status -w deployment/deployment-frontend

Waiting for deployment "deployment-frontend" rollout to finish: 1
out of 2 new replicas have been updated...
Waiting for deployment "deployment-frontend" rollout to finish: 1
out of 2 new replicas have been updated...
Waiting for deployment "deployment-frontend" rollout to finish: 1
out of 2 new replicas have been updated...
Waiting for deployment "deployment-frontend" rollout to finish: 1
old replicas are pending termination...
Waiting for deployment "deployment-frontend" rollout to finish: 1
old replicas are pending termination...
deployment "deployment-frontend" successfully rolled out
```

Now the deployment's rollout history:

```
$ kubectl rollout history deployment/deployment-frontend
deployment.extensions/deployment-frontend
```

```
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
```

And our application has a US Map page now:



New map page deployed

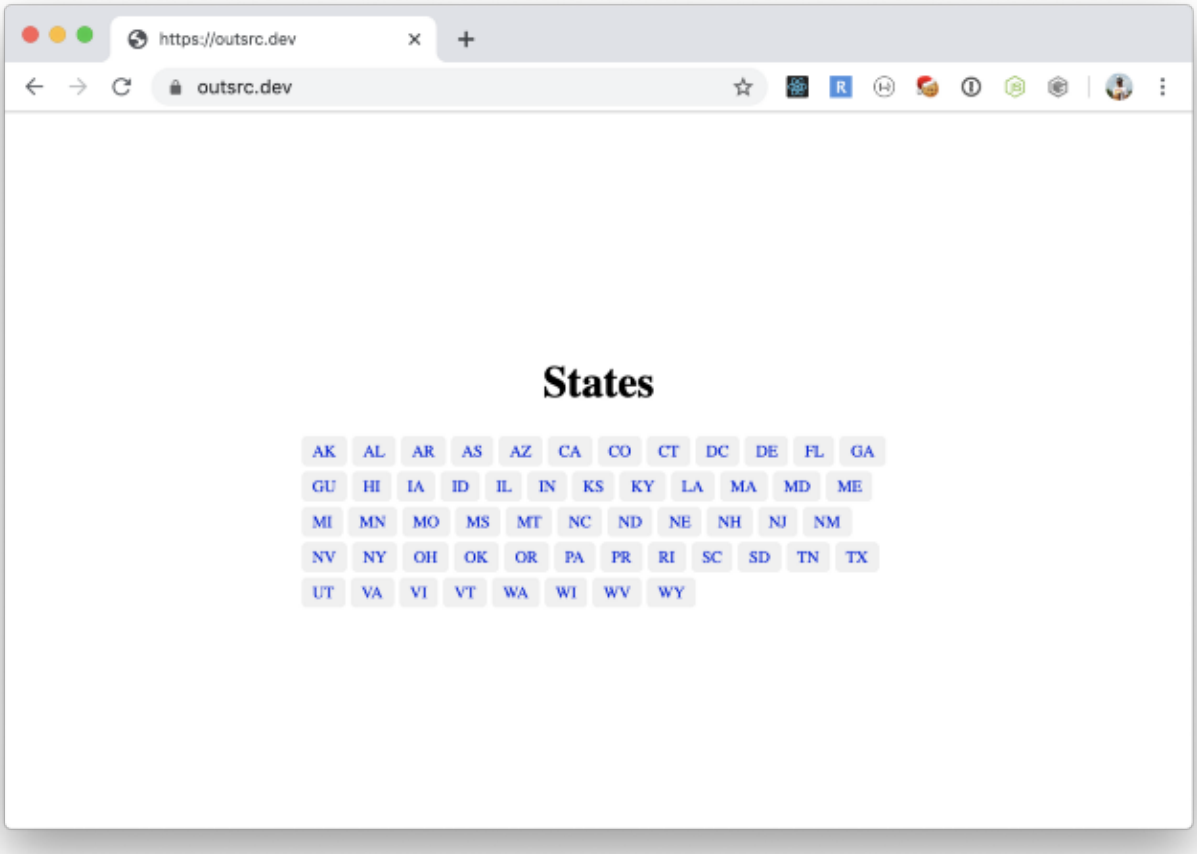
Oh no, we have a bug! rollback..

A user found a bug on our newly deployed version of the application. We need to rollback to the previous known working version.

Let's roll it back:

```
$ kubectl rollout undo deployment/deployment-frontend
deployment.extensions/deployment-frontend rolled back

$ kubectl rollout status -w deployment/deployment-frontend
Waiting for deployment "deployment-frontend" rollout to finish: 1
old replicas are pending termination...
Waiting for deployment "deployment-frontend" rollout to finish: 1
old replicas are pending termination...
deployment "deployment-frontend" successfully rolled out
```

Our application was reverted to previous version.



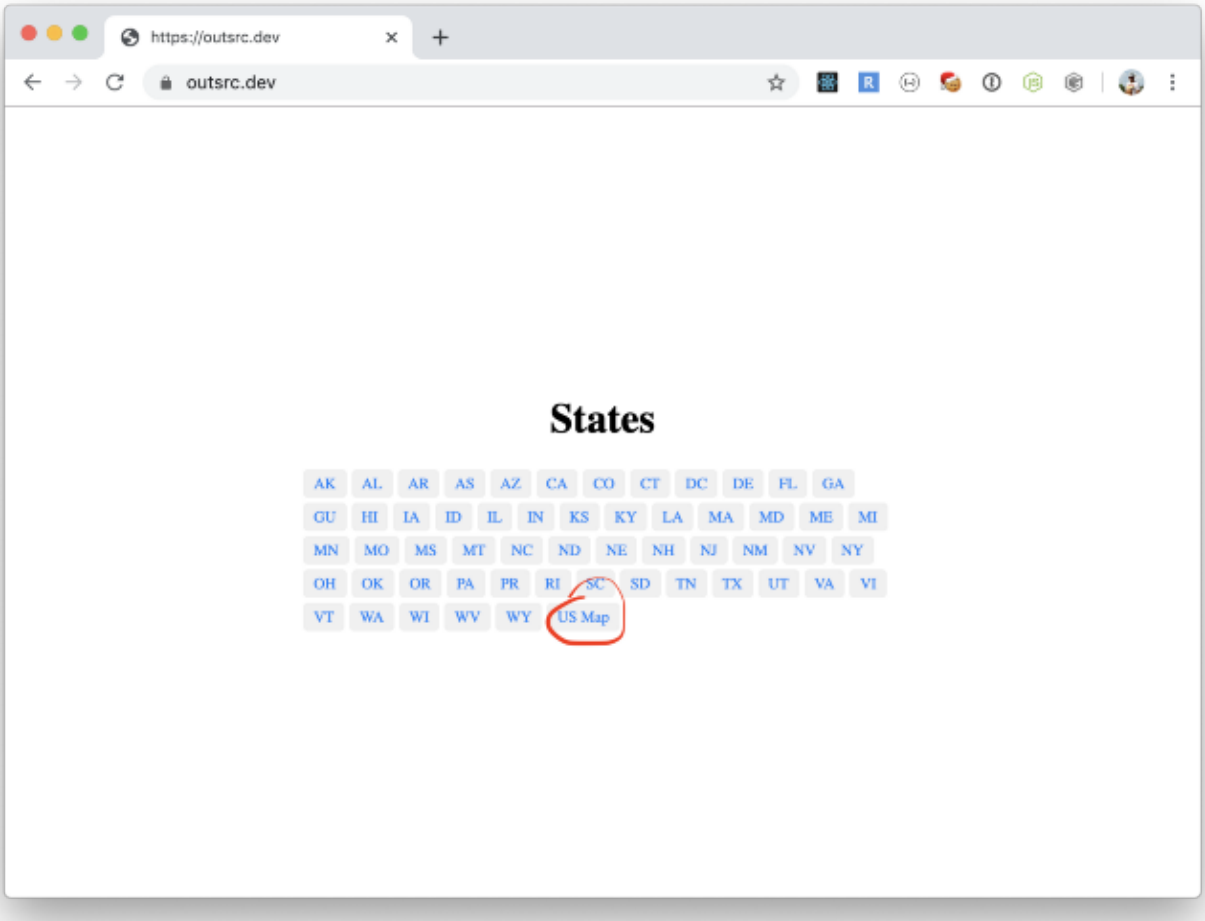
Application version 1.0.0

Notice on the history the revision numbers are still going up:

```
$ kubectl rollout history deployment/deployment-frontend
deployment.extensions/deployment-frontend
REVISION  CHANGE-CAUSE
2         <none>
3         <none>
```

We can always revert to any revision. In this case Revision #2 is our Maps revision. Let's bring it back.

```
$ kubectl rollout undo deployment/deployment-frontend --to-
revision=2
deployment.extensions/deployment-frontend rolled back

$ kubectl rollout status -w deployment/deployment-frontend
Waiting for deployment "deployment-frontend" rollout to finish: 1
old replicas are pending termination...
Waiting for deployment "deployment-frontend" rollout to finish: 1
old replicas are pending termination...
deployment "deployment-frontend" successfully rolled out
```



Back to version 1.1.0 (with US Map)

So far so good. Now, this going back and forth on a live website for features and bugs is not a good thing. Our users will feel frustrated if we roll out a feature just to find it has bugs and then roll it back. That's one of the reasons we have different **deployment environments.**

Is very common to find this set of environment:

master | staging | production

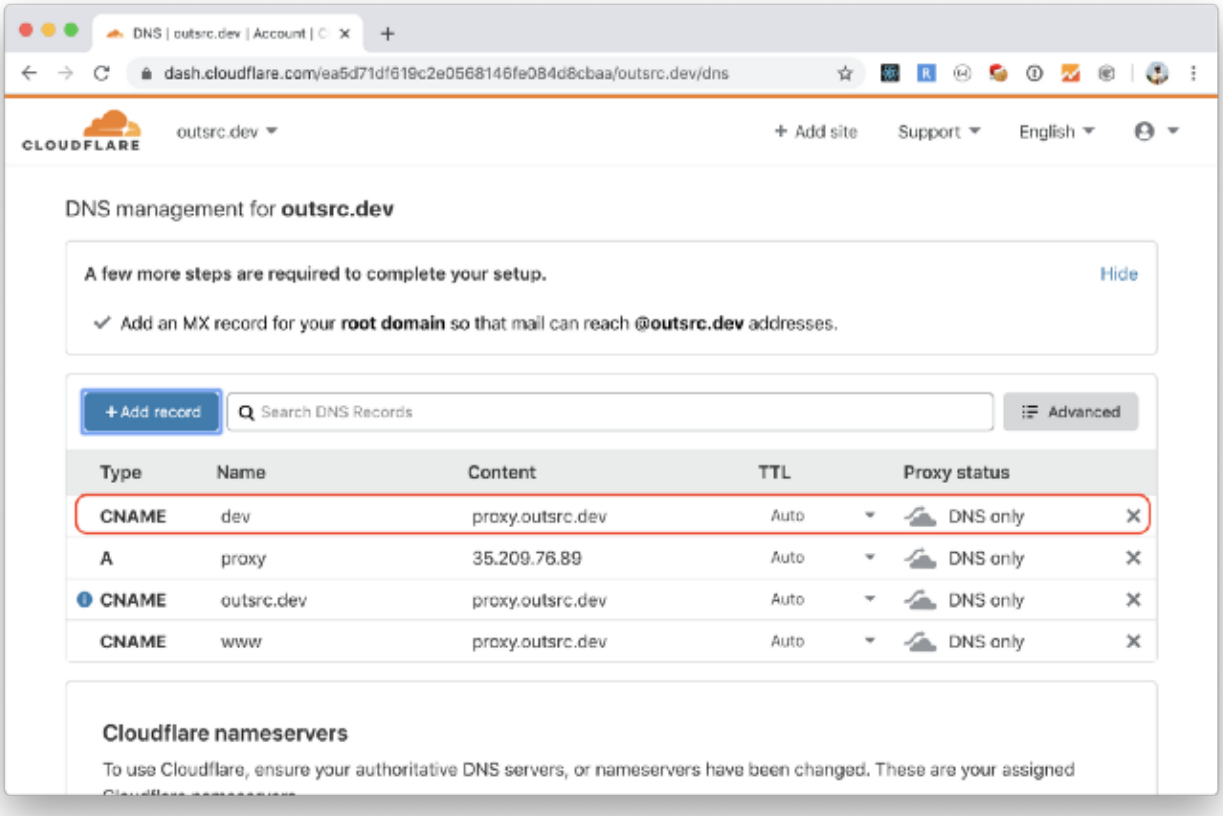
- **master**: Or development, usually most updated version, matches the master branch on the repo.
- **staging**: Most close to production, usually where last QA is performed.
- **production**: Is what your users see.

In this setting, any new feature or bugfix will go from master, to staging and then to production.

Out our US States application let's create one more environment: **development**

First we need to select a subdomain, for my current application I will choose: `dev.outsrc.dev` (master.outsrc.dev is fine too)

First: DNS, let's make a DNS registry making our subdomain pointing to the cluster proxy IP this is: (Remember our DNS from previous post was hosted on Cloudflare)



dev.outsrc.dev -> CNAME proxy.outsrc.dev

After this a NSLOOKUP of `dev.outsrc.dev` will return:

```
$ dig dev.outsrc.dev
; <<>> DiG 9.10.6 <<>> dev.outsrc.dev
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 38625
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;dev.outsrc.dev.   IN A


;; ANSWER SECTION:
dev.outsrc.dev.  300 IN CNAME proxy.outsrc.dev.
proxy.outsrc.dev. 300 IN A 35.209.76.89


;; Query time: 66 msec
;; SERVER: 192.168.1.254#53(192.168.1.254)
;; WHEN: Sun Dec 15 14:48:42 EST 2019
;; MSG SIZE  rcvd: 79
```

`dev.outsrc.dev` points right at our cluster.

Now Lets create a different set of resource files, on a different namespace: `outsrc-dev`

```
apiVersion: v1
kind: Namespace
metadata:
  name: outsrc-dev
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: dev-outsrc-dev-ingress
  namespace: outsrc-dev
  annotations:
    kubernetes.io/ingress.class: kong
    kubernetes.io/tls-acme: 'true'
    cert-manager.io/cluster-issuer: letsencrypt-production
spec:
  tls:
    - secretName: dev-outsrc-dev-tls
      hosts:
```

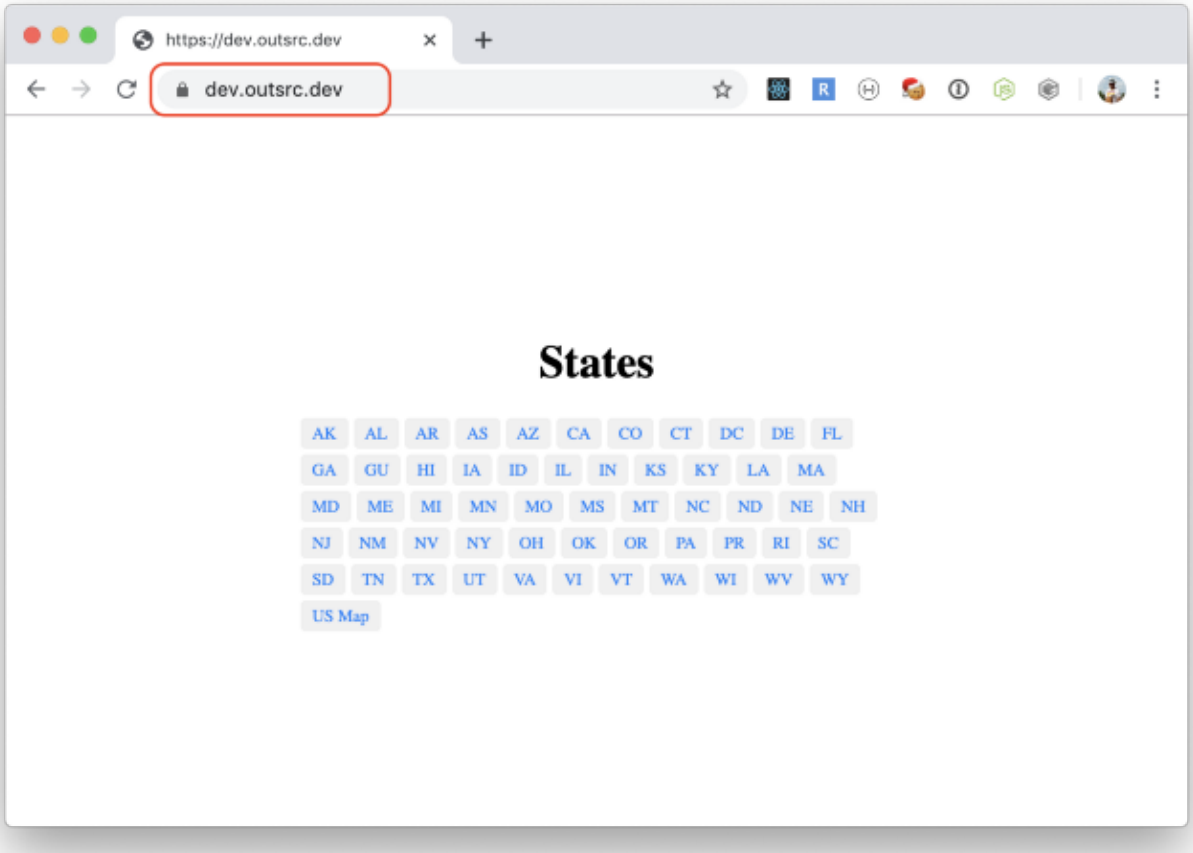
```
        - dev.outsrc.dev
  rules:
    - host: dev.outsrc.dev
      http:
        paths:
          - path: /api
            backend:
              serviceName: service-backend
              servicePort: 3000
          - path: /
            backend:
              serviceName: service-frontend
              servicePort: 3000
---
apiVersion: v1
kind: Service
metadata:
  name: service-frontend
  namespace: outsrc-dev
  labels:
    service: front
spec:
  selector:
    service: front
  ports:
    - port: 3000
      protocol: TCP
      targetPort: 3000
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-frontend
  namespace: outsrc-dev
  labels:
    service: front
spec:
  replicas: 2
  selector:
    matchLabels:
      service: front
  template:
    metadata:
      labels:
        service: front
    spec:
      containers:
        - name: frontend-container
```

```
          image: 'gcr.io/outsrc/outsrc-demo-front:1.1.0'
          imagePullPolicy: Always
          ports:
            - containerPort: 3000
          env:
            - name: API_URL
              value: 'https://outsrc.dev/api'
            - name: PORT
              value: '3000'
---
apiVersion: v1
kind: Service
metadata:
  name: service-backend
  namespace: outsrc-dev
  labels:
    service: back
spec:
  selector:
    service: back
  ports:
    - port: 3000
      protocol: TCP
      targetPort: 3000
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-backend
  namespace: outsrc-dev
  labels:
    service: back
spec:
  replicas: 2
  selector:
    matchLabels:
      service: back
  template:
    metadata:
      labels:
        service: back
    spec:
      containers:
        - name: backend-container
          image: 'gcr.io/outsrc/outsrc-demo-back:1.0.0'
          imagePullPolicy: Always
          ports:
```

```
114              - containerPort: 3000
115            env:
116              - name: PORT
117                value: '3000'
```

```
$ kubectl apply -f outsrc-dev.yml
namespace/outsrc-dev created
ingress.extensions/dev-outsrc-dev-ingress created
service/service-frontend created
deployment.apps/deployment-frontend created
service/service-backend created
deployment.apps/deployment-backend created
```

Ready, we now can access `http://dev.outsrc.dev`



dev.outsrc.dev

Now, this is a subdomain we don't want everybody to be able to access it. This is restricted to the internal developers team, the product managers and QA teams, designers, test engineers, etc.

We need to limit who can access this subdomain. There are several ways we can achieve this. One solution is using Kong Plugins (https://docs.konghq.com/hub/), more specific IP Restriction Plugin (https://docs.konghq.com/hub/kong-inc/ip-restriction/)

So now you know why I used Kong Ingress on the previous post.

Restrict Access by IP with Kong Plugin

First let's create a plugin resource descriptor

```yaml
apiVersion: configuration.konghq.com/v1
kind: KongPlugin
metadata:
  name: ip-restriction
  namespace: outsrc-dev
config:
  whitelist:
  - 202.110.224.38
  - 7.107.59.230
plugin: ip-restriction
```

ip-restrict.yml hosted with ❤ by **GitHub** **view raw**

Apply it first:

```
$ kubectl apply -f ip-restrict.yml
kongplugin.configuration.konghq.com/ip-restriction created
```

After this we can modify our Ingress resource file to signal that all routes should be IP restricted:

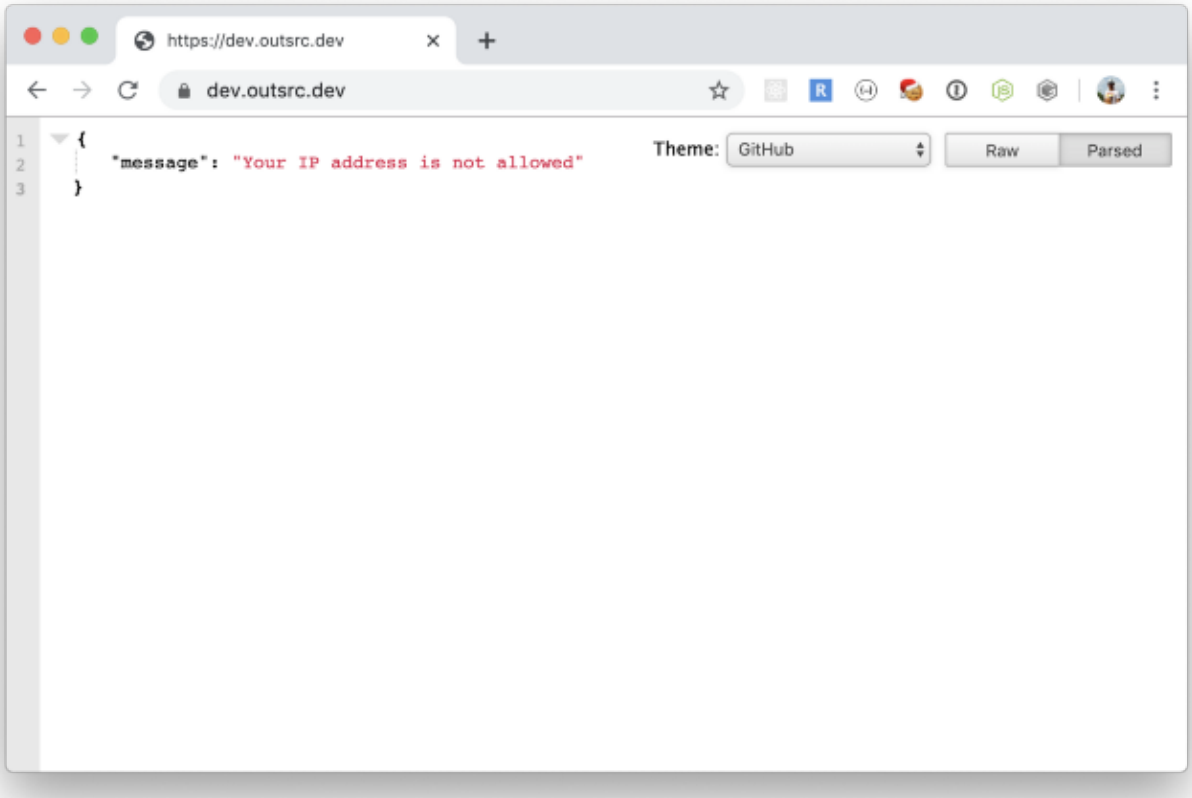
```
...
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: dev-outsrc-dev-ingress
  namespace: outsrc-dev
  annotations:
    kubernetes.io/ingress.class: kong
    kubernetes.io/tls-acme: 'true'
    cert-manager.io/cluster-issuer: letsencrypt-production
    plugins.konghq.com: ip-restriction
spec:
```

```
  tls:
    - secretName: dev-outsrc-dev-tls
      hosts:
        - dev.outsrc.dev
  rules:
    - host: dev.outsrc.dev
...
```

And update it:

```
$ kubectl apply -f outsrc-dev.yml
namespace/outsrc-dev unchanged
ingress.extensions/dev-outsrc-dev-ingress configured
service/service-frontend unchanged
deployment.apps/deployment-frontend unchanged
service/service-backend unchanged
deployment.apps/deployment-backend unchanged
```

Now if you try to access from an IP address not whitelisted you will be greeted by:



Denied!

Note: One nice thing about this IP restriction plugin is you can whitelist IP addresses and update only the plugin resource. The Ingress will use the updated list.

Conclusions

- Rolling updates require only a change on the deployment container.
- Rollbacks are actually quite fast.
- We can rollback to any previous deployed version. (Still not sure what are the limits here)
- To avoid going back and forth on production use deployment environments.
- Use different namespaces for your different environments if deploying on the same cluster.
- Optimally, create 2 clusters, separate Production environment from Staging and Development.
- Check Kong Plugins, some of them are really nice or you can write your own (Ready to learn Lua? or send me a text I might help you writing it)

Happy hacking…