Rolling Updates & Multiple Environments

Rolling Updates

Rolling Updates with Deployments

Rolling updates to pods can be done through updates of their respective deployment. If there are more than one replica in the deployment, a rolling update will be initiated. Kubernetes monitors the update and automatically stop a rollout if the health the new/updated pods fail.

How to update

There are several ways to update a Deployment.

- Edit the deployment manifest
  - locally/git + `kubectl apply -f <manifest-file>`
  - `kubectl edit deployment <deployment-name>`
- Set a new image
  - `kubectl set image deployment <deployment-name> <container-name>=<image>:<tag>`

Checking on updates

While the update is running you can check the status with

`kubectl rollout status deployment <deployment-name>`

Once it is done you can check the deployment:

`kubectl get deployment <deployment-name>`

`kubectl describe deployment <deployment-name>`

Deployment Audit Log

You can check the revisions of a deployment in the audit log (also called rollout history).

`kubectl rollout history deployment <deployment-name>`

For further details on what has actually changed in a specific revision

`kubectl rollout history deployment <deployment-name> --revision=2`

You can adjust the length of the history with `.spec.revisionHistoryLimit`.

Rolling Back

Rolling back to the previous version is easy.

```
kubectl rollout undo deployment <deployment-name>
```

You can also roll back to a specific former revision.

```
kubectl rollout undo deployment <deployment-name> --revision=2
```

Update Strategies

By default a rolling update strategy will be attached to your deployment. You can, however, also define the update strategy manually.

Your strategy type can be either `RollingUpdate` or `Recreate`. The latter kills all existing pods before creating new ones. The former does a rolling update with two variables that again can be manually defined:

- `maxUnavailable`
  - Defines the number (or percentage) of Pods that can be unavailable during the update process.
- `maxSurge`
  - Specifies the maximum number (or percentage) of Pods that can be created above the desired number of Pods.

By tweaking these variables you can adjust the rolling update strategy to your sepcific needs.

You can additionally sepcify a field called `minReadySeconds`, which defines the minimum number of seconds for which a newly created Pod should be ready without any of its containers crashing, for it to be considered available. By default this is set to `0`. Setting this to a higher number will slow down your update process, but might be useful to make sure your rollout is actually stable.

Blue/Green & Canary Deployments

You can work with `kubectl rollout pause` and `kubectl rollout resume`, however, that is not very sophisticated. The currently recommended way is to use a second Deployment (with usually fewer replicas), which is identical to the first Deployment, but has a different deployment name and updated image(s). As your pods are still labeled the same, your service will include them in its selector by default and route traffic to the new pods as soon as they're deployed.

Once you're confident that your new release is working as intended you can update your first Deployment and remove the second one again.

Further Reading

- Understanding Basic Kubernetes Concepts II - Using Deployments To Manage Your Services Declaratively
- Deployments Reference Documentation
- Managing Resources Reference Documentation

Multiple Environments

Multiple Environments & Isolation

When talking about multiple environments, there are many variables implied that need clarfication before choosing a fitting solution.

The first variable is the number of environment types needed, as in dev, testing, QA, pre-prod, prod, etc.

The second variable is if there are multiple teams and if these teams need separate environments.

However, these two variables only define the number of environments. On top of that the organization needs to decide on what level of isolation between environments is desired or in bigger organizations even required by compliance & security.

Separating Environments in Kubernetes

The most native way of separating environments in Kubernetes is using Namespaces and additionally specifiying nodes. However, there's also the option to use completely separate clusters if isolation requirements are high.

Isolation by Namespaces

Namespaces bring a certain level of isolation by default. They separate all resources created in them, but only on a soft level. That is, resources like for example pods are not blocked from using resources like for example services from other namespaces. They can access services in other namespaces through their FQDN on DNS or also through accessing the Kubernetes API.

If a real isolation is wanted it needs to be enforced with a combination of network policies as well as authorization and admission rules. All of these concepts are available in Kubernetes 1.4 already. However, some of them are still being actively worked on and might change and improve in future releases.

Isolation by Nodes

Additionally to namespace separation, the namespaces can be setup in a way to be using different nodes in the same cluster. This does not keep them from communicating or accessing objects in other namespaces

(which needs to be handled as mentioned above), but keeps evnironments from compeeting for the same physical reosurces like RAM and CPU.

Isolation by Clusters

The highest isolation is achieved by isolating when different environments result in different clusters. This way there is no interaction whatsoever between environments. However, several clusters need to be managed and keeping an overview might get complex. Further, as resources are not shared between clusters, there's potential for overhead in resource usage. Also, spinning up a new cluster is significantly slower than adding namespaces or even adding nodes.

Switching Namespaces

Switching namespaces is fairly straight-forward.

First, you can set the namespace for each `kubectl` command manually, by adding the `--namespace=<namespace-name>` to it. If you are using a number of namespaces regularly you can set aliases like the following:

```
alias kubectl-dev='kubectl --namespace=dev'
```

You can also change the default namespace of your `kubectl` configuration.

```
export CONTEXT=$(kubectl config view | awk '/current-context/ {print $2}')
kubectl config set-context $CONTEXT --namespace=<namespace-name>
```

Switching Clusters

Switching clusters is a bit different as each cluster has its own set of credentials or other kind of authorization mode.

Here you need to keep clusters as different contexts in your kubeconfig.

You can add and change context with `kubectl set context` and then use `kubectl config use-context <context-name>` to switch between clusters.

Using a federated set of clusters would be another option. However, federation is still in it's early days, and it softens the isolation between clusters a bit.

Keep Configuration and Secrets Out of Images and Deployments

To make your life easier and be able to deploy the same artifacts to different environments you should try to adhere the Config factor of the 12 factor app method. That means you should keep configuration and secrets out of your images and in Kubernetes also try to keep them out of your Deployments.

In Kubernetes you can use Config Maps and Secrets for that. Having separate Config Maps and Secrets for different environments will keep you from changing images between environments and make your life a lot easier.

Excourse Advanced Services

Services are a very powerful and flexible concept in Kubernetes. Especially, if you work with different environments.

Services abstract from your actual running Pods and containers, so you can have a fro example a DB Service with the same name in all your environments, but depending on the environment this DB Service points to a single DB instance, a DB cluster, or maybe in production even to an externally hosted managed DB like Amazon RDS.

As long as the interface (say SQL) stays the same, your other Pods that are consuming said DB Service will always talk to the "same" Service.

Further Reading

- Understanding Basic Kubernetes Concepts III - Services Give You Abstraction
- Understanding Basic Kubernetes Concepts IV - Secrets and ConfigMaps
- Namespaces Reference Documentation
- Config Maps Reference Documentation
- Secrets Reference Documentation
- Services Reference Documentation
- Deployments Reference Documentation
- Kubernetes Identity Management
- Federation
- Taints, Tolerations, and Dedicated Nodes