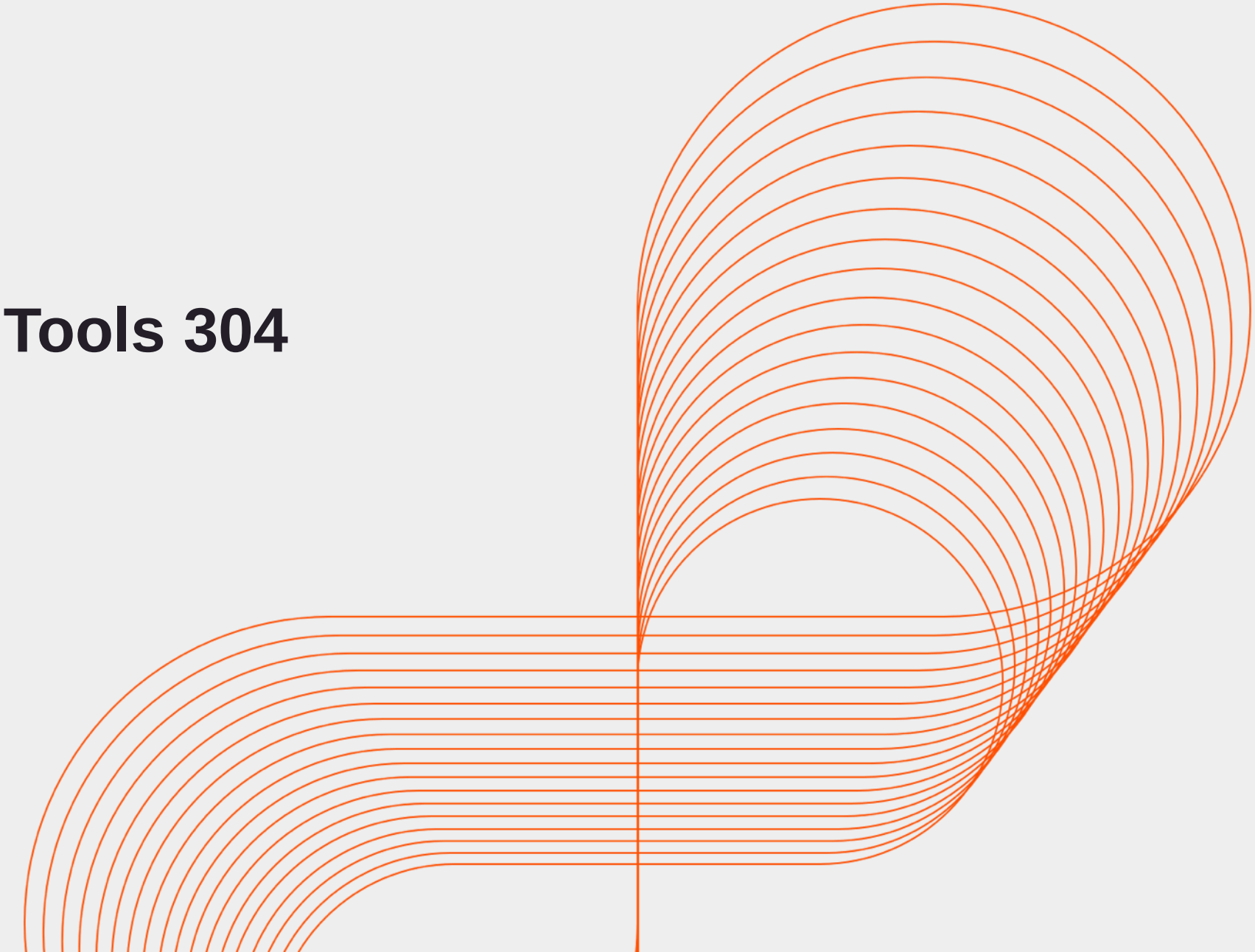
Persistent

Selfie Shots - Tools 304 Kubernetes

Persistent University



Deployments Overview

Key learning points :

1. API Access
2. Namespaces
3. Kubeconfig
4. API Maturity
5. Deployment Overview
6. Deployment Specs
7. Deployment Config

API Access

- Kubernetes has a powerful **REST**-based API. The entire architecture is API-driven. Knowing where to find resource endpoints and understanding how the API changes between versions can be important to ongoing administrative tasks, as there is much ongoing change and growth. Currently, there is no process for deprecation.
- As we learned in the *Architecture* chapter, the main agent for communication between cluster agents and from outside the cluster is the **kube-apiserve**r. A **curl** query to the agent will expose the current API groups. Groups may have multiple versions, which evolve independently of other groups, and follow a domain-name format with several names reserved, such as single-word domains, the empty group, and any name ending in **.k8s.io**.
- Such an API group is seen here:

```
$ curl https://127.0.0.1:6443/apis -k
....
   {
    "name": "apps",
    "versions": [
     {
      "groupVersion": "apps/v1beta1",
      "version": "v1beta1"
     },
     {
      "groupVersion": "apps/v1beta2",
      "version": "v1beta2"
     }
    ],
   },
....
```

Simple Pod

- As discussed earlier, a Pod is the lowest compute unit and individual object we can work with in Kubernetes. It can be a single container, but often, it will consist of a primary application container and one or more supporting containers.
- Below is an example of a simple pod manifest in YAML format. You can see the **apiVersion** (it must match the existing API group), the **kind** (the type of object to create), the **metadata** (at least a name), and its **spec** (what to create and parameters), which define the container that actually runs in this pod:

```
apiVersion: v1
kind: Pod
metadata:
    name: firstpod
spec:
    containers:
    - image: nginx
      name: stan
```

- You can use the kubectl create command to create this pod in Kubernetes. Once it is created, you can check its status with kubectl get pods. The output is omitted to save space:

```
$ kubectl create -f simple.yaml

$ kubectl get pods

$ kubectl get pod firstpod -o yaml

$ kubectl get pod firstpod -o json
```

kubeconfig

- The primary tool used from the command line will be **kubectl**, which calls **curl** on your behalf. You can also use the **curl** command from outside the cluster to view or make changes.
- The basic server information, with redacted TLS certificate information, can be found in the output of

$ kubectl config view

- If you view the verbose output from a previous page, you will note that the first line references a configuration file where this information is pulled from, ~/.kube/config

kubeconfig

- The output on the previous page shows multiple lines of output, with each of the keys being heavily truncated. While the keys may look similar, close examination shows them to be distinct:
- **apiVersion**
  As with other objects, this instructs the **kube-apiserver** where to assign the data.
- **clusters**
  This contains the name of the cluster, as well as where to send the API calls. The **certificate-authority-data** is passed to authenticate the **curl** request.
- **contexts**
  A setting which allows easy access to multiple clusters, possibly as various users, from one configuration file. It can be used to set **namespace**, **user**, and **cluster**.
- **current-context**
  Shows which cluster and user the **kubectl** command would use. These settings can also be passed on a per-command basis.
- **kind**
  Every object within Kubernetes must have this setting, in thus case a declaration of object type **Config**.
- **preferences**
  Currently not used, optional settings for the **kubectl** command, such as colorizing output.
- **users**
  A nickname associated with client credentials, which can be client key and certificate, username and password, and a token. Token and username/password are mutually exclusive. These can be configured via the **kubectl config set-credentials** command.

Namespaces

Every API call includes a namespace, using **default** if not otherwise declared: **https://10.128.0.3:6443/api/v1/namespaces/default/pods**.

Namespaces, a Linux kernel feature that segregates system resources, are intended to isolate multiple groups and the resources they have access to work with via quotas. Eventually, access control policies will work on namespace boundaries, as well. One could use *Labels* to group resources for administrative reasons. There are three namespaces when a cluster is first created.

- **default** This is where all resources are assumed, unless set otherwise.
- **kube-public** A namespace readable by all, even those not authenticated. General information is often included in this namespaces.
- **kube-system** Contains infrastructure pods.

Should you want to see all the resources on a system, you must pass the **--all-namespaces** option to the **kubectl** command.

Working with namespaces

Take a look at the following commands:

$ kubectl get ns

$ kubectl create ns test-ns

$ kubectl describe ns test-ns

$ kubectl get ns/test-ns -o yaml

$ kubectl delete ns/test-ns

- Once a namespace has been created, you can reference it via YAML when creating a resource:

```
$ cat redis.yaml
apiVersion: V1
kind: Pod
metadata:
    name: redis
    namespace: test-ns
```

API Resources with kubectl

- All API resources exposed are available via **kubectl**. To get more information, do

kubectl help.

kubectl [command] [type] [Name] [flag]

API Maturity

- The use of API groups and different versions allows for development to advance without changes to an existing group of APIs. This allows for easier growth and separation of work among separate teams. While there is an attempt to maintain some consistency between API and software versions, they are only indirectly linked.
- The use of JSON and Google's Protobuf serialization scheme will follow the same release guidelines.
- An *Alpha* level release, noted with *alpha* in the name, may be buggy and is disabled by default. Features could change or disappear at any time. Only use these features on a test cluster which is often rebuilt.
- The *Beta* level, found with *beta* in the name, has more well-tested code and is enabled by default. It also ensures that, as changes move forward, they will be tested for backwards compatibility between versions. It has not been adopted and tested enough to be called stable. You can expect some bugs and issues.
- Use of the *Stable* version, denoted by only an integer which may be preceded by the letter v, is for stable APIs.

Deployments

- The default controller for a container deployed via **kubectl run** is a *Deployment*.
- As with other objects, a *deployment* can be made from a YAML or JSON **spec** file. When added to the cluster, the controller will create a *ReplicaSet* and a *Pod* automatically. The containers, their settings and applications can be modified via an update, which generates a new *ReplicaSet*, which, in turn, generates new *Pods*.
- The updated objects can be staged to replace previous objects as a block or as a rolling update, which is determined as part of the deployment specification. Most updates can be configured by editing a YAML file and running **kubectl apply**. You can also use **kubectl edit** to modify the in-use configuration. Previous versions of the *ReplicaSets* are kept, allowing a rollback to return to a previous configuration.
- ***ReplicationControllers*** (RC) ensure that a specified number of pod replicas is running at any one time. *ReplicationControllers* also give you the ability to perform rolling updates. However, those updates are managed on the client side. This is problematic if the client loses connectivity, and can leave the cluster in an unplanned state. To avoid problems when scaling the RCs on the client side, a new resource has been introduced in the **extensions/v1beta1** API group: *Deployments*.
- Deployments allow server-side updates to pods at a specified rate. They are used for canary and other deployment patterns. Deployments generate *ReplicaSets*, which offer more selection features than *ReplicationControllers*, such as **matchExpressions**.

```
$ kubectl run dev-web --image=nginx:1.13.7-alpine
deployment "dev-web" created
```

Deployment Details

- In the previous slide, we created a new deployment running a particular version of the **nginx** web server.
- To generate the YAML file of the newly created objects, do:

$ kubectl get deployments,rs,pods -o yaml

- Sometimes, a JSON output can make it more clear:

$ kubectl get deployments,rs,pods -o json

- Now, we will look at the YAML output, which also shows default values, not passed to the object when created.

```
apiVersion: v1
items:
- apiVersion: extensions/v1beta1
  kind: Deployment
```

- **apiVersion**
  A value of **v1** shows that this object is considered to be a stable resource. In this case, it is not the *deployment*. It is a reference to the **List** type.

Deployment Details (Cntd)

- **items**
  As the previous line is a **List**, this declares the list of items the command is showing.
- **- apiVersion**
  The dash is a YAML indication of the first item, which declares the **apiVersion** of the object as **extensions/v1beta1**. While this would indicate that the object is not considered stable and will be dynamic, deployments are the suggested object to use.
- **kind**
  This is where the type of object to create is declared, in this case, a **deployment**.

Deployment Configuration Metadata

- Continuing with the YAML output, we see the next general block of output concerns the metadata of the deployment. This is where we would find labels, annotations, and other non-configuration information. Note that this output will not show all possible configuration. Many settings which are set to false by default are not shown, like **podAffinity** or **nodeAffinity**.

```
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  creationTimestamp: 2017-12-21T13:57:07Z
  generation: 1
  labels:
    run: dev-web
  name: dev-web
  namespace: default
  resourceVersion: "774003"
  selfLink: /apis/extensions/v1beta1/namespaces/default/deployments/dev-web
  uid: d52d3a63-e656-11e7-9319-42010a800003
```

Deployment Configuration Metadata

- Next, you can see an explanation of the information present in the deployment metadata (the file provided on the previous page):
- **annotations:** These values do not configure the object, but provide further information that could be helpful to third-party applications or administrative tracking. Unlike labels, they cannot be used to select an object with **kubectl**.
- **creationTimestamp :** Shows when the object was originally created. Does not update if the object is edited.
- **generation** : How many times this object has been edited, such as changing the number of replicas, for example.
- **labels** : Arbitrary strings used to select or exclude objects for use with **kubectl**, or other API calls. Helpful for administrators to select objects outside of typical object boundaries.
- **name** : This is a **required** string, which we passed from the command line. The name must be unique to the namespace.
- **resourceVersion** : A value tied to the **etcd** database to help with concurrency of objects. Any changes to the database will cause this number to change.
- **selfLink** : References how the **kube-apiserver** will ingest this information into the API.
- **uid** : Remains a unique ID for the life of the object.

Deployment Configuration Spec

- There are two **spec** declarations for the deployment. The first will modify the ReplicaSet created, while the second will pass along the Pod configuration.

```
spec:
  replicas: 1
  selector:
    matchLabels:
      run: dev-web
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
```

Deployment Configuration Spec

- **spec** : A declaration that the following items will configure the object being created.
- **replicas** : As the object being created is a *ReplicaSet*, this parameter determines how many Pods should be created. If you were to use **kubectl edit** and change this value to two, a second Pod would be generated.
- **selector** : A collection of values ANDed together. All must be satisfied for the replica to match. Do not create Pods which match these selectors, as the deployment controller may try to control the resource, leading to issues.
- **matchLabels** : Set-based requirements of the Pod selector. Often found with the **matchExpressions** statement, to further designate where the resource should be scheduled.
- **strategy** : A header for values having to do with updating Pods. Works with the later listed type. Could also be set to **Recreate**, which would delete all existing pods before new pods are created. With **RollingUpdate**, you can control how many Pods are deleted at a time with the following parameters.
- **maxsurge** : Maximum number of Pods over desired number of Pods to create. Can be a percentage, default of 25%, or an absolute number. This creates a certain number of new Pods before deleting old ones, for continued access.
- **maxUnavailable**: A number or percentage of Pods which can be in a state other than **Ready** during the update process.
- **type** : Even though listed last in the section, due to the level of white space indentation, it is read as the type of object being configured. (e.g. **RollingUpdate**).

Deployment Configuration Pod Template

- **template** : Data being passed to the *ReplicaSet* to determine how to deploy an object (in this case, containers).
- **containers** : Key word indicating that the following items of this indentation are for a container.
- **image** : This is the image name passed to the container engine, typically **Docker**. The engine will pull the image and create the Pod.
- **imagePullPolicy** : Policy settings passed along to the container engine, about when and if an image should be downloaded or used from a local cache.
- **name** : The leading stub of the Pod names. A unique string will be appended.
- **resources** : By default, empty. This is where you would set resource restrictions and settings, such as a limit on CPU or memory for the containers.
- **terminationMessagePath** : A customizable location of where to output success or failure information of a container.
- **terminationMessagePolicy** : The default value is **File**, which holds the termination method. It could also be set to **FallbackToLogsOnError** which will use the last chunk of container log if the message file is empty and the container shows an error.

Deployment Configuration Pod Template (Cntd)

- **dnsPolicy** : Determines if DNS queries should go to **kube-dns** or, if set to **Default**, use the node's DNS resolution configuration.
- **restartPolicy** : Used should the container be restarted if killed. Automatic restarts are part of the typical strength of Kubernetes.
- **scheduleName** : Allows for the use of a custom scheduler, instead of the Kubernetes default.
- **securityContext** : Flexible setting to pass one or more security settings, such as SELinux context, AppArmor values, users and UIDs for the containers to use.
- **terminationGracePeriodSeconds** : The amount of time to wait for a **SIGTERM** to run until a **SIGKILL** is used to terminate the container.

Deployment Configuration Status

- The **Status** output is generated when the information is requested:

```
status:
  availableReplicas: 2
  conditions:
  - lastTransitionTime: 2017-12-21T13:57:07Z
    lastUpdateTime: 2017-12-21T13:57:07Z
    message: Deployment has minimum availability.
    reason: MinimumReplicasAvailable
    status: "True"
    type: Available
  observedGeneration: 2
  readyReplicas: 2
  replicas: 2
  updatedReplicas: 2
```

- **availableReplicas** : Indicates how many were configured by the *ReplicaSet*. This would be compared to the later value of **readyReplicas**, which would be used to determine if all replicas have been fully generated and without error.
- **observedGeneration** : Shows how often the deployment has been updated. This information can be used to understand the rollout and rollback situation of the deployment.

Summary

At the end of this session, we see that you are now able to

- Understand the Kubernetes REST API
- Understand namespaces
- Understand a simple Pod template.
- Understand Deployment spec

Lab Exercise

- Create deployments with following options
  - Different replication factors
  - Update replica factors
  - Add/remove containers in Pod spec