### introduction

Hello, my name is Jeremy Cowan and I'm a Specialist SA for containers.  Today I'm going to show you how to give a demonstration of EKS that showcases a handful of the basic Kubernetes concepts.  But before we get to that, I want to give you a quick overview of Kubernetes and EKS. **[next]**

Kubernetes is an open source container orchestration system.  The role of a container orchestration system is to inventory the resources available within a cluster of hosts, schedule containers onto those hosts, monitor the health of those containers, and restart them when they fail. **[next]**

Kubernetes has become immensely popular with developers because it includes broad array of primitives for building cloud native applications, such ConfigMaps which can be use to store configuration information that applications can retrieve at run time; Secrets which can be used to store credentials for accessing other systems like databases; Persistent Volumes for dynamically provisioning storage volumes that can be mounted inside a container; and Jobs which are meant for executing containers a certain number of times which can be useful for running things like batch jobs.  All the these features are accessible through the Kubernetes API server.  Users typically interact with this API through a command line utility called kubectl which you'll see me use in the demo a little later. **[next]**

Another reason for Kubernetes popularity is its extensibility.  The Kubernetes API can be extended by creating Custom Resource Definitions or CRDs and Controllers.  A CRD is a way to store and retrieve structured data from Kubernetes whereas a Controller creates a declarative API that allows you to declare the desired state of a resource and tries to align the actual state with the desired state. **[next]**

All of the things that I've mentioned so far can be expressed as yaml manifest files that can be applied to a Kubernetes cluster.  A manifest is similar to a Cloudformation template insofar as it contains a declarative set of instructions that Kubernetes interprets to implement your desired state.

### Why we built EKS

Like a lot of our services, we built EKS in response to customer feedback.  Customers who were running Kubernetes on EC2 told us that building and maintaining a highly available, secure, Kubernetes cluster was hard.**[next]**

Achieving high availability often meant they had to deploy multiple instances of the Kubernetes masters in different availability zones.  The Kubernetes masters run several essential services such as the API server which I've already talked about, the scheduler, which is responsible for scheduling containers onto instances within the cluster, and the controller manager which is responsible for implementing and maintaining the desired state within the cluster.  Customers also had to deploy a highly available etcd database which serves as the backing store for Kubernetes, implement TLS, and integrate Kubernetes with an identity provider for authentication.  These customers overwhelmingly said they'd rather use a managed service rather than continue managing their own clusters.

Another reason we created EKS was “choice”.  Although we have our own container orchestration system known as the Elastic Container Service or ECS, we felt it was important to give customers the choice of running Kubernetes as a managed service.  We also felt that we could provide a differentiated offering by seamlessly integrating Kubernetes with other AWS services.  Much of this work is still ongoing. **[next]**

### The 3 tenants

When we were building EKS we adhered to the following tenants:

First, that the service would provide features and management capabilities to allow enterprises to run real workloads at real scale.  Reliability, visibility, scalability, and ease of management were our topmost priorities.  For instance, our control plane consists of at least 2 masters and 3 instances of etcd deployed across 3 availability zones to help ensure that the service is highly available. **[next]**

Second, EKS provides a native and upstream Kubernetes experience. Any modifications or improvements that we make in our service will be transparent to the Kubernetes community. This also means that your existing applications and investments in Kubernetes will work right out of the box with EKS. **[next]**

And third, EKS customers will not be forced to use additional AWS services, unless they want to; the integrations will be seamless and eliminate undifferentiated heavy lifting.  We are focused on making contributions to projects that allow customers to use the AWS components they currently know and love with their applications in Kubernetes. **[next]**

### Major differences between EKS and other managed Kubernetes offerings

So far, there are 2 places where we've integrated EKS with other AWS services.  The first is around our use of IAM for authentication.  Since Kubernetes doesn't maintain its own identity store it has to be configured to use an authentication module.  Kubernetes can use X509 client certificates, bearer tokens, an authenticating proxy, or HTTP basic auth to authenticate API requests through authentication plugins. EKS makes use of webhook authentication for verifying bearer tokens.  When a client like kubectl attempts to authenticate with the Kubernetes API server using a bearer token, an authentication webhook, which runs on the Kubernetes masters, POSTs a JSON-serialized object containing the token to IAM.  IAM then returns a response to the API that indicates the success or failure of the login.  Authorization, or the ability to consume Kubernetes APIs, is controlled by configuring Kubernetes RBAC roles.  With EKS, IAM becomes the de facto mechanism customers use to control access to Kubernetes API. **[next]**

The other place where we've integrated Kubernetes with AWS is around networking.  Rather than use an overlay network for pods, EKS makes use of a VPC container network interface (CNI) plugin that dynamically assigns pods an IP address from VPC's CIDR range.  This allows EKS to support very large clusters with minimal impact on performance.  It also allows customers to use services like VPC flow logs to clearly see see the traffic that is flowing into and out of each pod.  When the plugin starts it binds an ENI to the instance it's running on and allocates a pool of secondary IP addresses to that ENI.  When pods are started on the instance, an IPAM module that runs as part of the plugin, assigns it an IP address from the pool.  When the pod is stopped, the IP address is reclaimed. **[next]**

### demo time

Now that I've given you a brief overview of EKS, I'd like to show you how you can use the client kubectl to interact with the Kubernetes API.  In this demo, I will be creating a pod and exposing that pod as a service so that it is accessible from the Internet.  I will then show you how you can rollback from a bad deployment and use a ConfigMap to inject data into a pod at runtime.  Next, I will show you how you can run and login to the Kubernetes dashboard using your IAM credentials.  And finally, I will show you how to deploy an application through Helm, a package manager for deploying applications on Kubernetes.  Let's begin.

I'll start by create a pod.  A pod is a container or a related set of containers that are deployed as a unit.  Here I'm creating a pod that is based on the container, stenote/nginx-hostname, which runs an nginx server that displays the name of the host that it's running on.  In this case the host name is the name Kubernetes assigns to the pod.  You'll see that I've appended the command with the **record** flag.  This tells Kubernetes to keep track of the deployments of this pod **[run step1.sh]**.

kubectl run nginx --image=stenote/nginx-hostname --record=true

When you use the **run** command as I've done here, Kubernetes automatically creates a Deployment and a ReplicaSet.  A ReplicaSet tells Kubernetes how many replicas of this pod to run whereas a Deployment informs Kubernetes the rate at which to move pods from 1 ReplicaSet to the next as new versions are deployed.  Besides controlling the deployment rate, a deployment object also tracks the state of a deployment and maintains a history of deployments.  You can see that Kubernetes created both a deployment and a replicaset for the nginx container I ran in the previous step **[next]**.

kubectl get deployments,rs

Now I want to scale my application by deploying a second instance of the nginx pod.  To do that I'm going to use the command **scale** with the **replicas** flag to indicate how many replicas of this pod I want to run **[next]**.

kubectl scale deployment/nginx --replicas=2

As you can see, Kubernetes is now running 2 instances of my pod **[next]**.

kubectl get pods

The next thing I want to do is expose my application so that it's accessible from the Internet.  In order to do this I use the **expose** command along with the flag type=LoadBalancer.  This tells Kubernetes to create a Service of type Loadbalancer which automatically provisions an AWS classic load balancer and configures it forward traffic to your service.  By default, all of the instances within the cluster are registered as eligible targets.  When a worker node receives traffic from the Load Balancer, a service called the kube-proxy service will redirect the traffic to the appropriate pod within the cluster **[next]**.

kubectl expose deployment nginx --port 80 --target-port 80 --type=LoadBalancer --name=nginx-http

If I describe the service using the **kubectl describe service** command, I can see the DNS name of the classic load balancer that Kubernetes created when I exposed my service **[next]**.

kubectl describe service nginx-http

Next I'm going to curl the load balancer, but instead of copying and pasting the name of the load balancer, I'm going to use a Go template to extract the name of the load balancer from Kubernetes.  And as you can see, the service returns the name of the host that the container is running on **[next]**.

**[manually type or cut/paste]**

$ curl $(kubectl get svc/nginx-http -o go-template --template='{{(index .status.loadBalancer.ingress 0).hostname}}')

Next, I'm going to show you how you can use Deployments to rollback after a bad release.  The first thing I'm going to do it update the image for the deployment from stenote/nginx-hostname to nginx:1.91 **[next]**.

**[manually type or cut/paste]**

$ kubectl set image deployment/nginx nginx=nginx:1.91

Since there's no image with that tag, the deployment will fail.  To monitor the progress of the deployment I can use the **rollout status** command **[next]**.

**[manually type or cut/paste]**

$ kubectl rollout status deployment nginx

Let's investigate why the deployment is not working **[control-c, run step2.sh]**.

kubectl get rs

As you can see I have 2 replica sets, but 1 of them shows that 0 pods are ready.

Now let's look at the pods **[next]**.

kubectl get pods

It looks like 2 pods are having an issue pulling the image.  When we look at the details of the deployment **[next]**

kubectl describe deployment nginx

we can see that Kubernetes scaled down the original deployment to 1 and then tried scaling the new version of the pod to 2.  Since there's an issue with my deployment, I want to roll back to an earlier version.  To see a history of the deployments, I'm going to use the **rollout history** command **[next]**.

kubectl rollout history deployment/nginx

As you can see, I have 2 deployments: my original deployment of stenote/nginx-hostname and the updated deployment that is using a non-existent image.  Let's rollback to the previous version of the deployment **[next]**.

kubectl rollout undo deployment/nginx

Now when we look at the rollout history we can see that the deployment has been rolled back to version 1 **[next]**.

kubectl describe deployment/nginx

Next, I'm going to show you how you can fetch data from a ConfigMap at runtime.  **[run step3.sh]**

Let's start by creating an nginx configuration file.  When this file is loaded into nginx, the server will display the message “Hello, world!” **[next][next]**.

mkdir /Users/$(whoami)/conf.d    
cat > /Users/$(whoami)/conf.d/nginx-basic.conf << 'EOF'   
server {  
    location / {  
        return 200 'Hello, world!';  
        add\_header Content-Type text/plain;  
    }  
}  
EOF

Now let's create a configmap from this file. **[next]**

kubectl create configmap basic-config --from-file=/Users/$(whoami)/conf.d

To see that the file has been uploaded as a configmap, let's get the configmap we created.  The name of the configmap is basic-config.  We will reference this name when we create our pod.  **[next][next]**

kubectl get configmap -o yaml

Now that we have our configmap loaded, let's create a pod that uses our configmap.  Here I'm creating a kubernetes manifest for a pod that is mounting the configmap as a volume **[next]**.

cat > hello-world.yaml << 'EOF'  
apiVersion: v1  
kind: Pod  
metadata:  
  name: basic-pod  
spec:  
  containers:  
  - name: basic  
    image: nginx  
    volumeMounts:  
    - name: basic-config  
      mountPath: /etc/nginx/conf.d  
  volumes:  
    - name: basic-config  
      configMap:  
        name: basic-config  
EOF

The volume is mounted at the path /etc/nginx/conf.d which is where nginx loads its configuration from. Let's apply our pod manifest to the cluster. **[next]**

kubectl create -f hello-world.yaml

Now when I get the pods running on the cluster, I can see that my hello-world pod is running.  But how do I know that the configmap is really mounted?  Let's **exec** into the container and find out. **[next]**

kubectl get pods

Kuberentes has a feature known as exec which allows you to start an interactive session with a running pod.  It's great for troubleshooting issues, but should not be used to make changes to the pod.  If a change has to be made to a container within a pod, it's better to update the container image and then redeploy the pod or replace the container by using the **kubectl patch** command **[next]**.

kubectl exec -i -t basic-pod -- bash

Now that I'm in the container, I'm going to **cat** the nginx-basic.conf file in the conf.d directory.  You can see that it's the file that we uploaded as part of our configmap. **[manually type]**

$ cat /etc/nginx/conf.d/nginx-basic.conf

**[control-p, control-q to exit]**

Now let's verify that it works.  To do this, I'm going to use a Kubernetes feature called port-forward that allows me to create a proxy tunnel between my machine and the pod I created.  With the way I've configured it, the proxy tunnel will forward traffic from 8080 on localhost to my pod.  Now when I open a browser and specify localhost:8080 as the url, I see the message, “Hello, world!” **[manually type]**

$ kubectl port-forward basic-pod 8080:80

**[open browser and goto http://localhost:8080]**

Next I'm going to show you how to run the Kubernetes dashboard and login to it using your IAM credentials.  The Kubernetes dashboard is a UI that allows you perform a lot of the functions I've shown you so far.  To load the dashboard, I'm going to apply a manifest from a URL. **[next]**

**[manually type or cut/paste]**

$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/master/src/deploy/recommended/kubernetes-dashboard.yaml

To login the dashboard I need to pass in a bearer token.  Rather than using a bearer token for a **ServiceAccount** which is a static credential, I'm going to use a bearer token from IAM which has a lifespan of 15 minutes.  To generate the token, I'm going to use the aws-iam-authenticator binary to get a token I can use to login to the dashboard.

**[manually type or cut/paste]**

$ aws-iam-authenticator token -i <your cluster name> | jq -r '.status.token'

#Example output from the previous token generator step

#Use your generated token for browser dashboard token

#when accessing the dashboard url below.

k8s-aws-v1.aHR0cHM6Ly9zdHMuYW1hem9uYXdzLmNvbS8\_QWN0aW9uPUdldENhbGxlcklkZW50aXR5JlZlcnNpb249MjAxMS0wNi0xNSZYLUFtei1BbGdvcml0aG09QVdTNC1ITUFDLVNIQTI1NiZYLUFtei1DcmVkZW50aWFsPUFLSUFJTENUTFFaSDdaSVJST1NBJTJGMjAxODA5MTYlMkZ1cy1lYXN0LTElMkZzdHMlMkZhd3M0X3JlcXVlc3QmWC1BbXotRGF0ZT0yMDE4MDkxNlQwNDA2MjRaJlgtQW16LUV4cGlyZXM9NjAmWC1BbXotU2lnbmVkSGVhZGVycz1ob3N0JTNCeC1rOHMtYXdzLWlkJlgtQW16LVNpZ25hdHVyZT1iZTM4MTEyOGJhNTI3ZmNmMDVmOTJmNGI5OTdmYTA1ZjJjMTk3NDAzYjRlNDA2ODUwMjQ3ZmJlMmIyMmM3ZTJj

To access the dashboard, I'm going to start a proxy session.

**[manually type]**

$ kubectl proxy

This automatically handles authentication with apiserver and makes the Dashboard available at the <http://localhost:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/>.  After I select 'token' and paste the token into the appropriate field, I can see an overview of all workloads running on my cluster.

Finally, I'm going to show you how you can use Helm to deploy whole applications onto a Kubernetes cluster.  A typical application running on Kubernetes may consist of multiple Kubernetes objects, for instance, pods, configmaps, secrets, persistent volumes, services and so forth.  And while you can create manifests for each of the aforementioned objects, manifests don't really allow you to override or specify your own values for certain parameters.  Helm templates or “charts” provide that flexibility and they can be versioned and shared relatively easily.  Helm also support features such as upgrades and rollbacks.  It's quickly becoming the de facto way to package and deploy applications on Kubernetes.  Helm consists of 2 components: the client, Helm, and a server-side component called Tiller.  I've already gone ahead and installed Helm and Tiller in my environment.  Now I'm going to use Helm to deploy Wordpress **[manually type]**.

$ helm install stable/wordpress

In a few moments the load balancer for the service will be ready to start accepting requests.  For this demo I'm using a static password for Wordpress and MariaDB, but by default the chart will generate a random password that is stored as a secret.

When following the NOTES output, for getting the URL of the wordpress website, use the green highlighted name and replace in the yellow highlighted command.

NOTES:

1. Get the WordPress URL:

NOTE: It may take a few minutes for the LoadBalancer IP to be available.

Watch the status with: 'kubectl get svc --namespace default -w solid-dingo-wordpress'

export SERVICE\_IP=$(kubectl get svc --namespace default solid-dingo-wordpress -o jsonpath='{.status.loadBalancer.ingress[0].ip}')

echo "WordPress URL: http://$SERVICE\_IP/"

echo "WordPress Admin URL: http://$SERVICE\_IP/admin"

2. Login with the following credentials to see your blog

echo Username: user

echo Password: $(kubectl get secret --namespace default solid-dingo-wordpress -o jsonpath="{.data.wordpress-password}" | base64 --decode)

**[manually type]**

$ kubectl get svc --namespace default <solid-dingo-wordpress> -o jsonpath='{.status.loadBalancer.ingress[0]}'

#Output: This is the URL for the wordpress application. Paste into browser addr bar.

map[hostname:add0e8834b96711e8ac720a841b7016a-129856443.us-west-2.elb.amazonaws.com]

When I browse to the URL for service I see my Wordpress application.  (it may take 2-3 mins for the Wordpress application to activate)

### conclusion

That concludes our well-demonstrated video for EKS during which I showed you how to use Kuberbetes API to quickly deploy applications onto an EKS cluster, rollback from a failed deployment, expose an application to the internet, and load configuration information from a configmap.  I also went through how to login to the dashboard using your IAM credentials and deploying an application with Helm.  We welcome feedback about other ideas for future videos related to EKS and Kubenetes.  Thanks for watching.