A Real Application

In this section we will

- Work with a non-trivial application
- Create a Deployment
- Deploy the application on your cluster
- Scale the application
- Create a Service
- Expose the application on your cluster

Our Demo Application

This application is composed of multiple pieces:

- One main back-end service.
- A front-end (UI) service.
- A data layer.

We will deploy these pieces one at a time on our cluster.

Demo application

Clone the demo application's repository to your VM

```
$ git clone https://github.com/idcrosby/k8s-example.git
```

Recap of Resource Hierarchy

Deployment

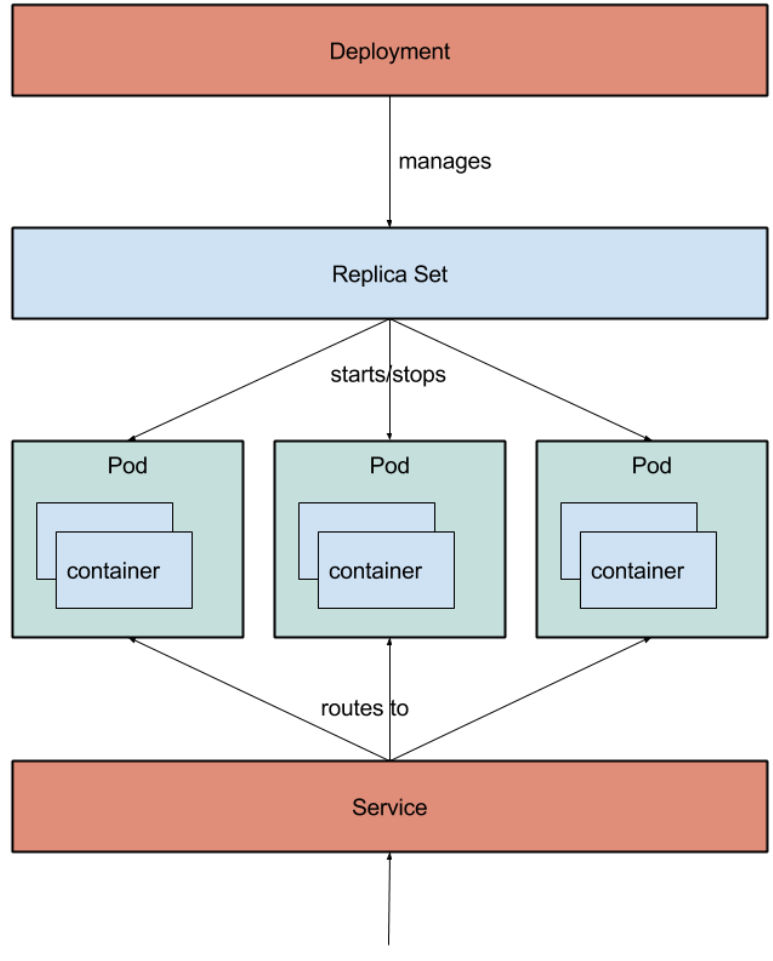
A Deployment manages ReplicaSets and defines how updates to Pods should be rolled out.

ReplicaSet

A ReplicaSet ensures that a specified number of Pods are running at any given time.

Pod

A Pod is a group of one or more containers deployed and scheduled together.


Deployment
manages
Replica Set
starts/stops
Pod
container
Pod
container
Pod
container
routes to
Service

Deployment Configuration

Look in "./resources" folder for the following Deployment configuration.

```
# resources/deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: k8s-real-demo
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: k8s-real-demo
    spec:
      containers:
      - name: k8s-real-demo
        image: icrosby/k8s-real-demo
        ports:
```

Deploy to the Cluster

```
$ kubectl apply -f resources/deployment.yaml
```

View Resource Details

Use the "kubectl get" and "kubectl describe" to view details of the deployed resources:

```
$ kubectl get deployments
$ kubectl get replicasets
$ kubectl get pods
```

```
$ kubectl describe pods <pod-name>
```

Interact with a Pod remotely

- Pods get a private IP address by default.
- Cannot be reached from outside the cluster.
- Use `kubectl port-forward` to map a local port to a port inside the `k8s-real-demo` pod.

Open 2 Terminals

Terminal 1

```
$ kubectl port-forward <pod-name> 8080:8080
```

Terminal 2

```
$ curl 0.0.0.0:8080

Hello from Container Solutions.
I'm running version 1.0 on k8s-real-demo-648d67845-hh8bn
```

Scaling Deployments

- Deployments manage ReplicaSets.
- Each deployment is mapped to one active ReplicaSet.
- Use `kubectl get replicasets` to view the current set of replicas.
- `kubectl get deployments` will give us the same info (plus more)

```
$ kubectl get rs

NAME                    DESIRED   CURRENT   READY     AGE
k8s-real-demo-364036756   1         1         1         16s
```

Scale up/down the Deployment

```
$ kubectl scale deployments k8s-real-demo --replicas=2

deployment "k8s-real-demo" scaled
```

Check the status of the Deployment

Notice the new Pod(s)

```
$ kubectl get pods
```

Look at the `Events` at the bottom

```
$ kubectl describe deployment k8s-real-demo
```

Fault Tolerance

What happens if we kill one of the Pods?

```
$ kubectl get pods
$ kubectl delete pod <pod-name>
```

Debugging

View the logs of a Pod

Use `kubectl logs` to view the logs for the `<pod-name>` Pod:

```
$ kubectl logs <pod-name>
```

Use the -f flag and observe what happens.

Run an interactive shell inside a Pod

Execute a shell in a Pod, like in Docker:

```
$ kubectl exec -ti <pod-name> /bin/sh
```

Accessing our Application

Reminder

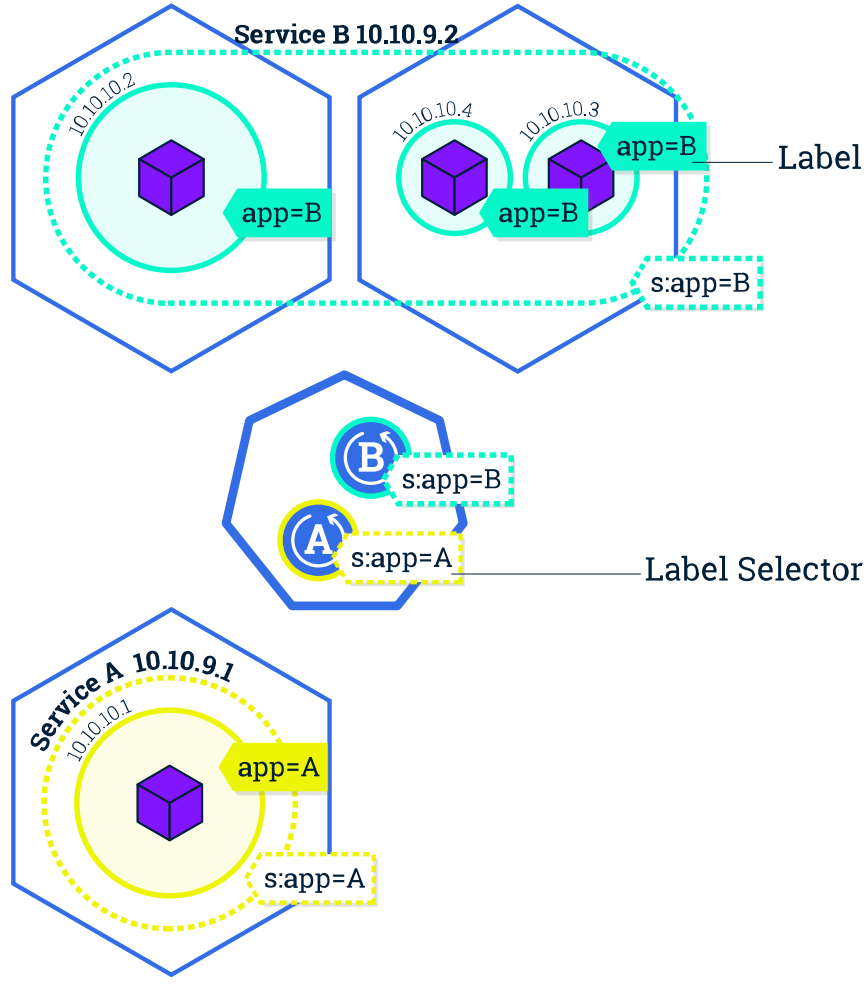
- Pods are ephemeral (no fixed IP)
- Port-forwarding strictly a debugging tool
- Need to be able to scale

Services

- Stable endpoints for Pods.
- Based on Labels and Selectors.

Labels & Selectors

- Label: key/value pair attached to objects (e.g. Pods)
- Selector: Identify and group a set of objects.


Service B 10.10.9.2
10.10.10.2
app=B
10.10.10.4
10.10.10.3
app=B
app=B
Label
s:app=B
B
s:app=B
A
s:app=A
Label Selector
Service A 10.10.9.1
10.10.10.1
app=A
s:app=A

Service Types

- ClusterIP (Default): Exposes the service on a cluster-internal IP.
- NodePort: Expose the service on a specific port on each node.
- LoadBalancer: Use a loadbalancer from a Cloud Provider. Creates `NodePort` and `ClusterIP`.
- ExternalName: Connect an external service (CNAME) to the cluster.

Service Configuration

Look in "./resources" folder for the following Service configuration.

```yaml
# resources/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: k8s-real-demo
  labels:
    app: k8s-real-demo
spec:
  type: NodePort
  ports:
  - name: http
    port: 80
    targetPort: 8080
  selector:
    app: k8s-real-demo
```

Create the Service

```
$ kubectl apply -f ./resources/service.yaml
```

Query the Service

Find the NodePort (via Service) and IP (via Node)

```
$ curl [IP]:[NODE_PORT]
```

Test Load Balancing

Make several calls to the service and notice the different responses.

Explore the Service

```
$ kubectl get services k8s-real-demo

NAME           CLUSTER-IP    EXTERNAL-IP    PORT(S)           AGE
k8s-real-demo   10.0.0.142    <nodes>        8080:30080/TCP    1m
```

```
$ kubectl describe services k8s-real-demo
```

Notice the `Endpoints:` entry

Labels

Using Labels

Use `kubectl get pods` with a label query, e.g. for troubleshooting.

```
$ kubectl get pods -l "app=k8s-real-demo"
```

Use `kubectl label` to add labels to a pod.

```
$ kubectl label pod [POD_NAME] 'secure=disabled'
```

```
$ kubectl get pods -l "app=k8s-real-demo"
$ kubectl get pods -l "secure=disabled"
```

We can also modify existing labels

```
$ kubectl label pod [POD_NAME] "app=new-label" --overwrite
$ kubectl describe pod [POD_NAME]
```

Endpoints

View the endpoints of the `k8s-real-demo` service:

(Note the difference from the last call to `describe`. What has happened?)

```
kubectl describe services k8s-real-demo
```

Revert the label to the original setting.

Deployments

Updating Deployments

(`RollingUpdate`)

- RollingUpdate is the default strategy.
- Updates Pods one (or a few) at a time.

Common Workflow

- Update the application, and create a new version.
- Build the new image and tag it with the new version, i.e. v2.
- Update the Deployment with the new image

Try It Out

First check the current version running

```
$ curl [EXTERNAL_IP]:[NodePort]

Hello from Container Solutions.
I'm running version 1.0 on k8s-real-demo-648d67845-jml8j
```

Next, update the image:

```
$ kubectl set image \
  deployment/k8s-real-demo k8s-real-demo=icrosby/k8s-real-demo:v2
```

Monitor the Deployment

Check status via

```
kubectl rollout status deployment k8s-real-demo
```

Now verify the new version

```
$ curl [EXTERNAL_IP]:[NodePort]
```

Now it's your turn

Step 1: Build your own Image

Build your own image and push to Docker Hub.

Open the example application in "k8s-example/"

```
docker build -t [DOCKERHUB_USER]/k8s-real-demo:v1.0.0 .
```

```
docker push [DOCKERHUB_USER]/k8s-real-demo:v1.0.0
```

Step 2: Create a Deployment

- Create a Deployment configuration for your Image.
  - Use the same Image name as above.
- Deploy on the cluster.

Step 3: Scale

- Scale the Deployment to 3 instances.
- Verify the scaling was successful and all instances are getting requests.

Step 4: Update

- Modify the `Dockerfile` to return a different Version.
- Build the Image and tag as `v2`.
- Update the Deployment to use the new tag.
- Verify the new version by making an HTTP request.
- View the logs of the application.

Step 5: Deploy the Front-end

In the /resources folder you will find configuration files for the front-end (Deployment and Service).

- ./resources/front-end-deploy.yaml
- ./resources/front-end-svc.yaml

Using these configuration files deploy and expose the application on to the cluster.

```
$ kubectl apply -f ./resources/front-end-deploy.yaml
```

```
$ kubectl apply -f ./resources/front-end-svc.yaml
```

Step 6: Accessing the Front-end

Find the port on which the front end is exposed (via the Service) and access this in your browser.

```
$ kubectl get svc front-end
```

Bonus Exercise

Storage

While we would like to ideally run stateless applications on Kubernetes, we will eventually run into the challenge of requiring state within our cluster.

CockroachDB

An open source 'Cloud Native' SQL database.

We will deploy CockroachDB to maintain the state for our demo application.

Deploying CockroachDB

```
$ kubectl apply -f https://raw.githubusercontent.com/cockroachdb/cock

service "cockroachdb-public" created
service "cockroachdb" created
poddisruptionbudget "cockroachdb-budget" unchanged
statefulset "cockroachdb" created
```

What did this do?

- Pull the Kubernetes configuration file from Github
- Created two Services
- Created a `poddisruptionbudget`
- Created a `StatefulSet`
  - Three Pods
  - Three PersistentVolumes
  - Three PersistentVolumeClaims

Don't worry, we will cover StatefulSets and PersistentVolumes later on

What have we Learned?

- How to deploy a 'real world' application on Kubernetes
- Deal with Deployment and Services
- Connecting Services with labels and selectors
- Scale up/down
- Update a Deployment (rolling update)

Next up, heading to Production...