Imperative versus declarative commands

Until now, our examples were focused on quick and imperative commands by using `kubectl <command>` i.e. to create deployments and services running on a one-node cluster with `minikube`. This is convenient for something quick, but does not easily expose the full flexibility of the API. To leverage all the options available via Kubernetes, it is often more effective to manage files that describe the deployment you want.

When using these files, you can use `kubectl <command>` along with the `-f option` to specify the file to use. Kubernetes offers a declarative mechanism as well, leveraging the `kubectl apply` command, which takes in files, reviews the current state, and manages the updates—creating, removing, and so on—as needed, while also keeping a simple audit log of the changes.

It's recommended using the declarative mechanism for anything more complex than running a single process, which will likely be most of your developed services. You may not need the audit trails in development. You probably would in a staging/canary environment or in production, so being familiar and comfortable with them is advantageous to understand them.

Best of all, you can include declarative files in source control, treating them like code. This gives you a consistent means of sharing that application structure among your team members, all of which can use it to provide a consistent environment.

The `kubectl apply command` has an -R option as well that will recursively descend directories if you are establishing a complex deployment.

What does declaration look like?

In most cases, the YAML `format` is used. The options and configurations can seem overwhelming. Each resource in Kubernetes has its own format, and some of those formats are changing and under active development. You will notice that some of the APIs and object structures will actively reference either **alpha** (means it is an early experiment and the data format may change) or **beta** (means is more solid than purely experimental, and can likely be used in production loads) to indicate the state of maturity of those resources in the project.
A great help is to get the declaration from existing Kubernetes objects. When you request the Kubernetes resource using the `kubectl get command` you can add the `-o yaml –export` option.

The `-o yaml` option could instead be `-o json` if you prefer that format. `–export` will strip out some extraneous information that is specific to the current state and identity of the resource within Kubernetes, and won't benefit you to store externally.

It's recommend using YAML as the format for these declarations. You could use JSON, but YAML allows you to add comments into the declarations, which are immensely useful for others reading those files.

Example 1: Deployment-Elements in YAML explained

```
apiVersion: apps/v1
kind: Deployment                 (1)
metadata:
  name: calculator-deployment    (2)
  labels:
    app: calculator
spec:
  replicas: 3                    (3)
  selector:                      (4)
    matchLabels:
      app: calculator
  template:                      (5)
    metadata:
      labels:                    (6)
        app: calculator
    spec:
      containers:
      - name: calculator         (7)
        image: gcr.io/calculator (8)
        ports:                   (9)
        - containerPort: 8080
```

In this YAML configuration, we have to ensure the following:

1. We have defined a Kubernetes resource of the Deployment type from the `apps/v1` Kubernetes API version.
2. The unique deployment name is `calculator-deployment`.
3. We have defined that there should be exactly 3 of the same Pods created.
4. `selector` defines how Deployment finds Pods to manage, in this case, just by the label.
5. `template` defines the specification for each created Pod.
6. Each Pod is labeled with `app: calculator`.
7. Each Pod contains a Docker container `named calculator`.
8. A Docker container was created from the image called `gcr.io/calculator`.
9. The Pod exposes container port `8080`.

To install the deployment, run the following command:

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

You can check that the three Pods, each containing one Docker container, have been created:

```
$ kubectl get pods
NAME                                  READY STATUS  RESTARTS AGE
calculator-deployment-dccdf8756-h2l6c 1/1   Running 0        1m
calculator-deployment-dccdf8756-tgw48 1/1   Running 0        1m
calculator-deployment-dccdf8756-vtwjz 1/1   Running 0        1m
```

Each Pod runs a Docker container. We can check its logs by using the following command:

```
$ kubectl logs pods/calculator-deployment-dccdf8756-h2l6c
```

Example 2: Kubernetes Deployment (YAML)

As an example, containerized python app is used. The parameter `terminationGracePeriodSeconds` is set to give your app enough time to finish work.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-python-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-python-app
  template:
    metadata:
      labels:
        app: my-python-app
    spec:
      terminationGracePeriodSeconds: 60
      containers:
        - name: app
          image: your-image:latest
          ports:
            - containerPort: 8080
```

Example 3: Kubernetes Deployment + Service (YAML)

The previous example declared only the deployment. In this example, we add the corresponding service.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-python-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-python-app
  template:
    metadata:
```

```
      labels:
        app: my-python-app
    spec:
      terminationGracePeriodSeconds: 60
      containers:
        - name: app
          image: your-image:latest
          ports:
            - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: my-python-service
spec:
  selector:
    app: my-python-app
  ports:
    - port: 8080
      targetPort: 8080
  type: ClusterIP
```

Service types

As a recapture on how your application can be accessed from the outside, we need to start with the types of Kubernetes Services. You can use four different service types, as follows:

- `ClusterIP (default)`: The service has an internal IP only.
- `NodePort`: Exposes the service on the same port of each cluster node. In other words, each physical machine (which is a Kubernetes node) opens a port that is forwarded to the service. Then, you can access it by using `<NODE-IP>:<NODE-PORT>`.
- `LoadBalancer`: Creates an external load balancer and assigns a separate external IP for the service. Your Kubernetes cluster must support external load balancers, which works fine in the case of cloud platforms, but may not work if you use minikube.
- `ExternalName`: Exposes the service using a DNS name (specified by externalName in the spec).

While LoadBalancer seems to be the simplest solution, it has two drawbacks:

- It's not always available, for example, if you deployed on-premises Kubernetes or used minikube.
- External public IPs are usually expensive. A different solution is to use a NodePort service.

```
$ kubectl get service my-python-service
NAME                TYPE     CLUSTER-IP    EXTERNAL-IP  PORT(S)        AGE
my-python-service  NodePort 10.19.248.154 <none>       8080:32259/TCP 13m
```

You can see that port 32259 was selected as a node port. This means that we can access our `my-python-service` service using that port and the IP of any of the Kubernetes nodes.

The IP address of your Kubernetes node depends on your installation. In the case of minikube, you can check it with the minikube ip command:

```
$ kubectl get nodes -o jsonpath='{
$.items[*].status.addresses[?(@.type=="ExternalIP")].address }'
35.192.180.252 35.232.125.195 104.198.131.248
```

To check that you can access `my-python-service` from the outside, run the following command:

```
$ curl http://<NODE-IP>:32047/
```

Based on „Kubernetes for Developers by Joseph Heck“