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 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
```