YAML Basics for Kubernetes

It’s difficult to escape YAML if you’re doing anything related to many software fields — particularly Kubernetes. YAML, which stands for Yet Another Markup Language, or YAML Ain’t Markup Language (depending who you ask) is a human-readable text-based format for specifying configuration-type information. For example, in this article, we’ll pick apart the YAML definitions for creating first a Pod, and then a Deployment.

Using YAML for K8s definitions gives you a number of advantages, including:

- **Convenience:** You’ll no longer have to add all of your parameters to the command line
- **Maintenance:** YAML files can be added to source control, so you can track changes
- **Flexibility:** You’ll be able to create much more complex structures using YAML than you can on the command line

YAML is a superset of JSON, which means that any valid JSON file is also a valid YAML file. So on the one hand, if you know JSON and you’re only ever going to write your own YAML (as opposed to reading other people’s) you’re all set. On the other hand, that’s not very likely, unfortunately. Even if you’re only trying to find examples on the web, they’re most likely in (non-JSON) YAML, so we might as well get used to it. Still, there may be situations where the JSON format is more convenient, so it’s good to know that it’s available to you.

Fortunately, there are only two types of structures you need to know about in YAML:

- Lists
- Maps

That’s it. You might have maps of lists and lists of maps, and so on, but if you’ve got those two structures down, you’re all set. That’s not to say there aren’t more complex things you can do, but in general, this is all you need to get started.

YAML Maps

Let’s start by looking at YAML maps. Maps let you associate name-value pairs, which of course is convenient when you’re trying to set up configuration information. For example, you might have a config file that starts like this:

```
---
apiVersion: v1
kind: Pod
```

The first line is a separator, and is optional unless you’re trying to define multiple structures in a single file. From there, as you can see, we have two values, v1 and Pod, mapped to two keys, apiVersion and kind.

This kind of thing is pretty simple, of course, and you can think of it in terms of its JSON equivalent:

```
{
   "apiVersion": "v1",
   "kind": "Pod"
}
```

Notice that in our YAML version, the quotation marks are optional; the processor can tell that you’re looking at a string based on the formatting.

You can also specify more complicated structures by creating a key that maps to another map, rather than a string, as in:

```
---
apiVersion: v1
kind: Pod
metadata:
  name: rss-site
  labels:
    app: web
```

In this case, we have a key, metadata, that has as its value a map with 2 more keys, name and labels. The labels key itself has a map as its value. You can nest these as far as you want to.

The YAML processor knows how all of these pieces relate to each other because we’ve indented the lines. In this example I’ve used 2 spaces for readability, but the number of spaces doesn’t matter — as long as it’s at least 1, and as long as you’re CONSISTENT. For example, name and labels are at the same indentation level, so the processor knows they’re both part of the same map; it knows that app is a value for labels because it’s indented further.

Quick note: NEVER use tabs in a YAML file.

So if we were to translate this to JSON, it would look like this:

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
               "name": "rss-site",
               "labels": {
                          "app": "web"
                         }
              }
}
```

Now let's look at lists.

YAML lists

YAML lists are literally a sequence of objects. For example:

```
args:
  - sleep
  - "1000"
  - message
  - "Bring back Firefly!"
```

As you can see here, you can have virtually any number of items in a list, which is defined as items that start with a dash (-) indented from the parent. So in JSON, this would be:

```
{
   "args": ["sleep", "1000", "message", "Bring back Firefly!"]
}
```

And of course, members of the list can also be maps:

```
---
apiVersion: v1
kind: Pod
metadata:
  name: rss-site
  labels:
    app: web
spec:
  containers:
    - name: front-end
      image: nginx
      ports:
        - containerPort: 80
    - name: rss-reader
      image: nickchase/rss-php-nginx:v1
      ports:
        - containerPort: 88
```

So as you can see here, we have a list of containers "objects", each of which consists of a name, an image, and a list of ports. Each list item under ports is itself a map that lists the containerPort and its value.

For completeness, let's quickly look at the JSON equivalent:

```
{
   "apiVersion": "v1",
   "kind": "Pod",
   "metadata": {
                 "name": "rss-site",
                 "labels": {
                             "app": "web"
                           }
               },
    "spec": {
       "containers": [{
                       "name": "front-end",
                       "image": "nginx",
                       "ports": [{
                                  "containerPort": "80"
                                 }]
                      },
                      {
                       "name": "rss-reader",
                       "image": "nickchase/rss-php-nginx:v1",
                       "ports": [{
                                  "containerPort": "88"
                                 }]
                      }]
            }
}
```

As you can see, we're starting to get pretty complex, and we haven't even gotten into anything particularly complicated! No wonder YAML is replacing JSON so fast.

So let's review. We have:

- maps, which are groups of name-value pairs
- lists, which are individual items
- maps of maps
- maps of lists
- lists of lists

- lists of maps

Basically, whatever structure you want to put together, you can do it with those two structures.

Creating a Pod using YAML

OK, so now that we've got the basics out of the way, let's look at putting this to use. We're going to first create a Pod, then a Deployment, using YAML.

If you haven't set up your cluster and kubectl, go ahead and check out this article series on setting up Kubernetes before you go on. It's OK, we'll wait….

Back already? Great! Let's start with a Pod.

Creating the pod file

In our previous example, we described a simple Pod using YAML:

```
—
apiVersion: v1
kind: Pod
metadata:
 name: rss-site
 labels:
   app: web
spec:
 containers:
   – name: front-end
     image: nginx
     ports:
       – containerPort: 80
   – name: rss-reader
     image: nickchase/rss-php-nginx:v1
     ports:
       – containerPort: 88
```

Taking it apart one piece at a time, we start with the API version; here it's just v1. (When we get to deployments, we'll have to specify a different version because Deployments don't exist in v1.)

Next, we're specifying that we want to create a Pod; we might specify instead a Deployment, Job, Service, and so on, depending on what we're trying to achieve.

Next we specify the metadata. Here we're specifying the name of the Pod, as well as the label we'll use to identify the pod to Kubernetes.

Finally, we'll specify the actual objects that make up the pod. The spec property includes any containers, storage volumes, or other pieces that Kubernetes needs to know about, as well as properties such as whether to restart the container if it fails. Let's take a closer look at a typical container definition:

```
...
 spec:
   containers:
     - name: front-end
       image: nginx
       ports:
         - containerPort: 80
     - name: rss-reader
 ...
```

In this case, we have a simple, fairly minimal definition: a name (front-end), the image on which it's based (nginx), and one port on which the container will listen internally (80). Of these, only the name is really required, but in general, if you want it to do anything useful, you'll need more information.

You can also specify more complex properties, such as a command to run when the container starts, arguments it should use, a working directory, or whether to pull a new copy of the image every time it's instantiated. You can also specify even deeper information, such as the location of the container's exit log. Here are the properties you can set for a Container:

- name
- image
- command
- args
- workingDir
- ports
- env
- resources
- volumeMounts
- livenessProbe
- readinessProbe
- lifecycle
- terminationMessagePath
- imagePullPolicy
- securityContext
- stdin

- stdinOnce
- tty

Now let’s go ahead and actually create the pod.

Creating the pod using the YAML file

The first step, of course, is to go ahead and create a text file.   Call it pod.yaml and add the following text, just as we specified it earlier:

```
---
 apiVersion: v1
 kind: Pod
 metadata:
   name: rss-site
   labels:
     app: web
 spec:
   containers:
     - name: front-end
       image: nginx
       ports:
         - containerPort: 80
     - name: rss-reader
       image: nickchase/rss-php-nginx:v1
       ports:
         - containerPort: 88
```

Save the file, and tell Kubernetes to create its contents:

```
> kubectl create -f pod.yaml
pod "rss-site" created
```

As you can see, K8s references the name we gave the Pod. You can see that if you ask for a list of the pods:

```
> kubectl get pods
 NAME       READY     STATUS              RESTARTS   AGE
 rss-site   0/2       ContainerCreating   0          6s
```

If you check early enough, you can see that the pod is still being created. After a few seconds, you should see the containers running:

```
> kubectl get pods
NAME       READY     STATUS    RESTARTS   AGE
rss-site   2/2       Running   0          14s
```

From here, you can test out the Pod (just as we did in the previous article), but ultimately we want to create a Deployment, so let’s go ahead and delete it so there aren’t any name conflicts:

```
> kubectl delete pod rss-site
pod "rss-site" deleted
```

Troubleshooting pod creation

Sometimes, of course, things don’t go as you expect. Maybe you’ve got a networking issue, or you’ve mistyped something in your YAML file. You might see an error like this:

```
> kubectl get pods
NAME       READY     STATUS         RESTARTS   AGE
rss-site   1/2       ErrImagePull   0          9s
```

In this case, we can see that one of our containers started up just fine, but there was a problem with the other. To track down the problem, we can ask Kubernetes for more information on the Pod:

```
> kubectl describe pod rss-site
Name:           rss-site
Namespace:      default
Node:           10.0.10.7/10.0.10.7
Start Time:     Sun, 08 Jan 2017 08:36:47 +0000
Labels:         app=web
Status:         Pending
IP:             10.200.18.2
Controllers:    <none>
Containers:
  front-end:
    Container ID:               docker://a42edaa6dfbfdf161f3df5bc6af05e740b97fd9ac3d35317a6dcda77b0310759
    Image:                      nginx
    Image ID:                   docker://sha256:01f818af747d88b4ebca7cdabd0c581e406e0e790be72678d257735fad84a15f
    Port:                       80/TCP
    State:                      Running
      Started:                  Sun, 08 Jan 2017 08:36:49 +0000
    Ready:                      True
    Restart Count:              0
    Environment Variables:      <none>
  rss-reader:
    Container ID:
    Image:                      nickchase/rss-php-nginx
    Image ID:
    Port:                       88/TCP
    State:                      Waiting
      Reason:                   ErrImagePull
    Ready:                      False
    Restart Count:              0
    Environment Variables:      <none>
Conditions:
  Type          Status
  Initialized   True
  Ready         False
  PodScheduled  True
No volumes.
QoS Tier:       BestEffort
Events:
  FirstSeen     LastSeen        Count   From                    SubobjectPath  Type             Reason
      Message
  ---------     --------        -----   ----                    -------------  --------
------          -------
  45s           45s             1       {default-scheduler
}                   Normal          Scheduled               Successfully assigned rss-site to 10.0.10.7
  44s           44s             1       {kubelet 10.0.10.7}     spec.containers{front-
end}      Normal          Pulling                 pulling image "nginx"
  45s           43s             2       {kubelet
10.0.10.7}                    Warning          MissingClusterDNS       kubelet does not have ClusterDNS IP
configured and cannot create Pod using "ClusterFirst" policy. Falling back to DNSDefault policy.
  43s           43s             1       {kubelet 10.0.10.7}     spec.containers{front-
end}      Normal          Pulled                  Successfully pulled image "nginx"
  43s           43s             1       {kubelet 10.0.10.7}     spec.containers{front-
end}      Normal          Created                 Created container with docker id a42edaa6dfbf
  43s           43s             1       {kubelet 10.0.10.7}     spec.containers{front-
end}      Normal          Started                 Started container with docker id a42edaa6dfbf
  43s           29s             2       {kubelet 10.0.10.7}     spec.containers{rss-
reader}     Normal          Pulling                 pulling image "nickchase/rss-php-nginx"
  42s           26s             2       {kubelet 10.0.10.7}     spec.containers{rss-
reader}     Warning         Failed                  Failed to pull image "nickchase/rss-php-nginx": Tag latest
not found in repository docker.io/nickchase/rss-php-nginx
  42s           26s             2       {kubelet
10.0.10.7}                    Warning          FailedSync              Error syncing pod, skipping: failed to
"StartContainer" for "rss-reader" with ErrImagePull: "Tag latest not found in repository docker.io/nickchase/rss-
php-nginx"


  41s   12s     2       {kubelet 10.0.10.7}     spec.containers{rss-reader}    Normal   BackOff         Back-off
pulling image "nickchase/rss-php-nginx"
  41s   12s     2       {kubelet 10.0.10.7}                                    Warning  FailedSync      Error
syncing pod, skipping: failed to "StartContainer" for "rss-reader" with ImagePullBackOff: "Back-off pulling image
\"nickchase/rss-php-nginx\""
```

As you can see, there's a lot of information here, but we're most interested in the Events — specifically, once the warnings and errors start showing up. From here I was able to quickly see that I'd forgotten to add the :v1 tag to my image, so it was looking for the :latest tag, which didn't exist.

To fix the problem, I first deleted the Pod, then fixed the YAML file and started again. Instead, I could have fixed the repo so that Kubernetes could find what it was looking for, and it would have continued on as though nothing had happened,.

Now that we've successfully gotten a Pod running, let's look at doing the same for a Deployment.

Creating a Deployment using YAML

Finally, we're down to creating the actual Deployment. Before we do that, though, it's worth understanding what it is we're actually doing.

K8s, remember, manages container-based resources. In the case of a Deployment, you're creating a set of resources to be managed. For example, where we created a single instance of the Pod in the previous example, we might create a Deployment to tell Kubernetes to manage a set of replicas of that Pod — literally, a ReplicaSet — to make sure that a certain number of them are always available. So we might start our Deployment definition like this:

```
---
 apiVersion: extensions/v1beta1
 kind: Deployment
 metadata:
   name: rss-site
 spec:
   replicas: 2
```

Here we're specifying the apiVersion as extensions/v1beta1 — remember, Deployments aren't in v1, as Pods were — and that we want a Deployment. Next we specify the name. We can also specify any other metadata we want, but let's keep things simple for now.

Finally, we get into the spec. In the Pod spec, we gave information about what actually went into the Pod; we'll do the same thing here with the Deployment. We'll start, in this case, by saying that whatever Pods we deploy, we always want to have 2 replicas. You can set this number however you like, of course, and you can also set properties such as the selector that defines the Pods affected by this Deployment, or the minimum number of seconds a pod must be up without any errors before it's considered "ready".

OK, so now that we know we want 2 replicas, we need to answer the question: "Replicas of what?" They're defined by templates:

```
---
 apiVersion: extensions/v1beta1
 kind: Deployment
 metadata:
   name: rss-site
 spec:
   replicas: 2
   template:
     metadata:
       labels:
         app: web
     spec:
       containers:
         - name: front-end
           image: nginx
           ports:
             - containerPort: 80
         - name: rss-reader
           image: nickchase/rss-php-nginx:v1
           ports:
             - containerPort: 88
```

Look familiar? It should; it's virtually identical to the Pod definition in the previous section, and that's by design. Templates are simply definitions of objects to be replicated — objects that might, in other circumstances, by created on their own.

Now let's go ahead and create the deployment. Add the YAML to a file called deployment.yaml and point Kubernetes at it:

```
> kubectl create -f deployment.yaml
deployment "rss-site" created
```

To see how it's doing, we can check on the deployments list:

```
> kubectl get deployments
NAME       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
rss-site   2         2         2            1           7s
```

As you can see, Kubernetes has started both replicas, but only one is available. You can check the event log by describing the Deployment, as before:

```
> kubectl describe deployment rss-site
Name:                   rss-site
Namespace:              default
CreationTimestamp:      Mon, 09 Jan 2017 17:42:14 +0000=
Labels:                 app=web
Selector:               app=web
Replicas:               2 updated | 2 total | 1 available | 1 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
OldReplicaSets:         <none>
NewReplicaSet:          rss-site-4056856218 (2/2 replicas created)
Events:
  FirstSeen     LastSeen        Count   From                            SubobjectPath   Type            Reason
            Message
  ---------     --------        -----   ----                            -------------   --------        ------
            -------
  46s           46s             1       {deployment-controller
}               Normal           ScalingReplicaSet       Scaled up replica set rss-site-4056856218 to 2
```

As you can see here, there’s no problem, it just hasn’t finished scaling up yet. Another few seconds, and we can see that both Pods are running:

```
> kubectl get deployments
NAME       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
rss-site   2         2         2            2           1m
```