# Practice M7: Templating Tools and Package Management

For the purpose of this practice, we will assume that we are working on a machine with either a Windows 10/11 or any recent Linux distribution and there is a local virtualization solution (like VirtualBox, Hyper-V, VMware Workstation, etc.) installed

***Please note that long commands may be hard to read here. To handle this, you can copy them to a plain text editor first. This will allow you to see them correctly. Then you can use them as intended***

*For this practice we will assume that we are working on a three-node cluster (one control plane node and two worker nodes) with an arbitrary pod network plugin installed*

## Part 1: Templating Tools

We will try a few out of the many available options

Let's start with the simplest and perhaps less scalable option – the manual approach

### Manual Approach

Check the following (**part1/1-manual/1-appa.yaml**) manifest file

apiVersion: apps/v1

kind: Deployment

metadata:

  name: appa

spec:

  replicas: 1

  selector:

    matchLabels:

      app: appa

  template:

    metadata:

      labels:

        app: appa

    spec:

      containers:

      - name: main

        image: shekeriev/k8s-environ:latest

        env:

        - name: APPROACH

          value: "STATIC"

        - name: FOCUSON

          value: "APPROACH"

---

apiVersion: v1

kind: Service

metadata:

  name: appa

spec:

  type: NodePort

  ports:

  - port: 80

    nodePort: 30001

    protocol: TCP

  selector:

    app: appa

It will spin up a deployment with one pod replica and a service of type **NodePort**

Wait, do not start it yet

What, if we want to spin it once with one image, then with another? Or with different versions (tags) of an image? Or exposed on different port?

Of course, we can go and edit the manifest every time, but this is not the way we should go

Another option would be to use placeholders on the positions we would want to easily change

The format of those placeholders is up to us

We can use for example ***%variable%*** and later look for it and replace it with an appropriate value

Check this (**part1/1-manual/2-appa.yaml**) manifest

apiVersion: apps/v1

kind: Deployment

metadata:

  name: appa

spec:

  replicas: %replicas%

  selector:

    matchLabels:

      app: appa

  template:

    metadata:

      labels:

        app: appa

    spec:

      containers:

      - name: main

        image: %image%:%tag%

        env:

        - name: APPROACH

          value: "%approach%"

        - name: FOCUSON

          value: "APPROACH"

---

apiVersion: v1

kind: Service

metadata:

  name: appa

spec:

  type: NodePort

  ports:

  - port: 80

    nodePort: %nodeport%

    protocol: TCP

  selector:

    app: appa

If we send it as it is to the cluster, we will see that it will result in error as it is not valid

But having it prepared in this way, we can change or adjust it on the fly

We can execute this command to change, for example ***the number of replicas***

**sed s/%replicas%/3/ part1/1-manual/2-appa.yaml**

This way, we set just ***one placeholder*** and saw the new version of the manifest

In order to set all of them, we must either execute a sequence of commands *(skip this)*

**sed s/%replicas%/3/ part1/1-manual/2-appa.yaml | sed s@%image%@shekeriev/k8s-environ@ | sed s/%tag%/latest/ | sed s/%approach%/MANUAL/ | sed s/%nodeport%/30001/**

Or execute an extended *(this is one of the possible ways)* command *(use this)*

**sed 's/%replicas%/3/ ; s@%image%@shekeriev/k8s-environ@ ; s/%tag%/latest/ ; s/%approach%/MANUAL/ ; s/%nodeport%/30001/' part1/1-manual/2-appa.yaml**

Here *(in either of two versions)*, there are **two things** to notice

**First**, we are replacing ***the first occurrence*** of a placeholder. Should we want to replace all, we must change the rules. For example, instead of this ***s/%nodeport%/30001/*** we should have this ***s/%nodeport%/30001/g***

**Second**, notice how we changed the separator in one of the rules. We did it because we have the same character as part of the value that we want to use

We can either use the above command *(either of its versions)* to store the new version of the manifest *(skip this for now)*

**sed 's/%replicas%/3/ ; s@%image%@shekeriev/k8s-environ@ ; s/%tag%/latest/ ; s/%approach%/MANUAL/ ; s/%nodeport%/30001/' part1/1-manual/2-appa.yaml > part1/1-manual/3-appa.yaml**

Or just send it to the cluster *(use this approach)*

**sed 's/%replicas%/3/ ; s@%image%@shekeriev/k8s-environ@ ; s/%tag%/latest/ ; s/%approach%/MANUAL/ ; s/%nodeport%/30001/' part1/1-manual/2-appa.yaml | kubectl apply -f -**

Open a browser tab and navigate to the following address http://<control-plane-ip>:30001

It is working 😊

Now, let's change the values of the placeholders with the following command

**sed 's/%replicas%/3/ ; s@%image%@shekeriev/k8s-environ@ ; s/%tag%/green/ ; s/%approach%/MANUAL/ ; s/%nodeport%/30001/' part1/1-manual/2-appa.yaml | kubectl apply -f -**

Return to the browser tab and refresh

It is still working but looking a little bit different

So, this appears to be a valid approach of templating

There is one drawback though, we must submit values for all placeholders before we can use the manifest even if we want to change just one

Perhaps, we can think of some clever way to tackle this, but this is not the point here

Let's clean up and try something else

**kubectl delete deployment appa**

**kubectl delete service appa**

### Kustomize

#### Install

Under any **Linux** distribution, we can download it *(the latest version)* with

**curl -s "https://raw.githubusercontent.com/kubernetes-sigs/kustomize/master/hack/install\_kustomize.sh" | bash**

For other versions or operating systems, we should check here:

<https://github.com/kubernetes-sigs/kustomize/releases>

Once downloaded, we must make it executable and move it to a folder that is part of our executable path

For example, execute

**sudo mv kustomize /usr/local/bin**

We can check if the process went as expected by executing

**kustomize version**

#### Hello World

Let's follow one of the examples from the official repository to get on speed

You can find the samples here: <https://github.com/kubernetes-sigs/kustomize/tree/master/examples>

Prepare a set of folders, for example in the **/tmp** folder

**DEMO=/tmp/hello**

**BASE=$DEMO/base**

**mkdir -p $BASE**

Now, download the files

**curl -s -o "$BASE/#1.yaml" "https://raw.githubusercontent.com\**

**/kubernetes-sigs/kustomize\**

**/master/examples/helloWorld\**

**/{configMap,deployment,kustomization,service}.yaml"**

Check the folders and files so far

**tree $DEMO**

Explore the content of the files *(change the service to be* ***NodePort****)* but leave the **kustomization.yaml** for last

The rest of the files are just like the ones, we worked so far

**kustomization.yaml** contains the base structure of our customizable application

**cat $BASE/kustomization.yaml**

Then, we can invoke

**kustomize build $BASE**

To see where we are starting from

In a similar way, we can test the application by executing

**kustomize build $BASE | kubectl apply -f -**

Check the assigned port

**kubectl get services**

And open a browser tab and navigate to http://<control-plane-ip>:<node-port>

The application is working *(we see its basic version, before the adjustments)*

Now, delete it by executing

**kustomize build $BASE | kubectl delete -f -**

Let's do a basic customization by editing the **$BASE/kustomization.yaml** file

**sed -i.bak 's/app: hello/app: my-hello/' $BASE/kustomization.yaml**

We can see the effect with

**kustomize build $BASE**

All instances of the app label have changed from **hello** to **my-hello**

It is something, but this is not what we wanted to see

Let's introduce the **overlays**

First, create a set of additional folders

**OVERLAYS=$DEMO/overlays**

**mkdir -p $OVERLAYS/staging**

**mkdir -p $OVERLAYS/production**

Check what we have so far in terms of folders

**tree $DEMO**

We have created a new folder *(****overlays****)* which will store our customizations

There, we created two folders *(one for each customization)* – **staging** and **production**

Each folder will contain a set of files that describe the customizations

Create the main customization file for the staging

**cat <<'EOF' >$OVERLAYS/staging/kustomization.yaml**

**namePrefix: staging-**

**labels:**

**- includeSelectors: true**

**pairs:**

**org: acmeCorporation**

**variant: staging**

**commonAnnotations:**

**note: Hello, I am staging!**

**resources:**

**- ../../base**

**patches:**

**- path: map.yaml**

**EOF**

And then the patch file for the staging that will change the configuration map of the base

**cat <<EOF >$OVERLAYS/staging/map.yaml**

**apiVersion: v1**

**kind: ConfigMap**

**metadata:**

**name: the-map**

**data:**

**altGreeting: "Have a pineapple!"**

**enableRisky: "true"**

**EOF**

Then, add the main customization file for production

**cat <<EOF >$OVERLAYS/production/kustomization.yaml**

**namePrefix: production-**

**labels:**

**- includeSelectors: true**

**pairs:**

**org: acmeCorporation**

**variant: production**

**commonAnnotations:**

**note: Hello, I am production!**

**resources:**

**- ../../base**

**patches:**

**- path: deployment.yaml**

**EOF**

And finally, the production patch which will increase the number of replicas *(from 3 to 5)*

**cat <<EOF >$OVERLAYS/production/deployment.yaml**

**apiVersion: apps/v1**

**kind: Deployment**

**metadata:**

**name: the-deployment**

**spec:**

**replicas: 5**

**EOF**

Now, we can check again the hierarchy that we created

**tree $DEMO**

By now, we should have something like this

**/tmp/hello**

**├── base**

**│   ├── configMap.yaml**

**│   ├── deployment.yaml**

**│   ├── kustomization.yaml**

**│   ├── kustomization.yaml.bak**

**│   └── service.yaml**

**└── overlays**

**├── production**

**│   ├── deployment.yaml**

**│   └── kustomization.yaml**

**└── staging**

**├── kustomization.yaml**

**└── map.yaml**

**4 directories, 9 files**

And then, compare the manifests generated for both environments

**diff <(kustomize build $OVERLAYS/staging) <(kustomize build $OVERLAYS/production) | more**

We can see each individual set of files. Check first the ones for the staging

**kustomize build $OVERLAYS/staging**

And then for production

**kustomize build $OVERLAYS/production**

Now, spin up the staging variant

**kustomize build $OVERLAYS/staging | kubectl apply -f -**

*Or we can go directly with*

***kubectl apply -k $OVERLAYS/staging***

Check the port given to the service

**kubectl get services**

And open a browser tab and navigate to http://<control-plane-ip>:<node-port>

The application is working *(we see its staging version, after the adjustments)*

Now, spin up also its production version

**kustomize build $OVERLAYS/production | kubectl apply -f -**

*Or we can go directly with*

***kubectl apply -k $OVERLAYS/production***

Check the port given to the service

**kubectl get services**

And open a browser tab and navigate to http://<control-plane-ip>:<node-port>

The application is working *(we see its production version, after the adjustments)*

Now, we have two copies of our application deployed and working on our cluster

We can explore their components and see the differences

Then, to delete them, we must execute

**kustomize build $OVERLAYS/staging | kubectl delete -f -**

**kustomize build $OVERLAYS/production | kubectl delete -f -**

*Or we can go directly with*

***kubectl delete -k $OVERLAYS/staging***

***kubectl delete -k $OVERLAYS/production***

#### Our Own Scenario

Now, let's combine what we just saw with our application from earlier

Go to folder **part1/2-kustomize**

Check the starting set of folders and the only file (**base/appa.yaml**)

First, let's create the base customization file (**base/kustomization.yaml**)

**apiVersion: kustomize.config.k8s.io/v1beta1**

**kind: Kustomization**

**resources:**

**- appa.yaml**

Check that everything is working

**kustomize build base/**

Then create the overlays folders

**mkdir -p overlays/{blue,green}**

Now, add customization file for the blue variant (**overlays/blue/kustomization.yaml**)

**apiVersion: kustomize.config.k8s.io/v1beta1**

**kind: Kustomization**

**namePrefix: blue-**

**labels:**

**- includeSelectors: true**

**pairs:**

**variant: blue**

**resources:**

**- ../../base**

**patches:**

**- path: custom-env.yaml**

**- path: custom-rs.yaml**

It sets custom environment variable value and replica count

And then for the green variant (**overlays/green/kustomization.yaml**)

**apiVersion: kustomize.config.k8s.io/v1beta1**

**kind: Kustomization**

**namePrefix: green-**

**labels:**

**- includeSelectors: true**

**pairs:**

**variant: green**

**resources:**

**- ../../base**

**patches:**

**- path: custom-env.yaml**

**- path: custom-np.yaml**

It sets custom environment variable value and service node port

Now, it is time for the patch files

We will start with the set for the **blue** variant

First, the environment patch (**overlays/blue/custom-env.yaml**)

**apiVersion: apps/v1**

**kind: Deployment**

**metadata:**

**name: appa**

**spec:**

**template:**

**spec:**

**containers:**

**- name: main**

**env:**

**- name: APPROACH**

**value: "KUSTOMIZE (BLUE)"**

And then the replica count patch (**overlays/blue/custom-rs.yaml**)

**apiVersion: apps/v1**

**kind: Deployment**

**metadata:**

**name: appa**

**spec:**

**replicas: 5**

Now, let's prepare the set for the **green** variant

First, the environment patch (**overlays/green/custom-env.yaml**)

**apiVersion: apps/v1**

**kind: Deployment**

**metadata:**

**name: appa**

**spec:**

**template:**

**spec:**

**containers:**

**- name: main**

**env:**

**- name: APPROACH**

**value: "KUSTOMIZE (GREEN)"**

And then the service node port patch (**overlays/green/custom-np.yaml**)

**apiVersion: v1**

**kind: Service**

**metadata:**

**name: appa**

**spec:**

**ports:**

**- nodePort: 30002**

**port: 80**

**protocol: TCP**

Okay, we are almost ready

Let's change the image (or at least the tag) for each of the two variants

Enter the **blue** folder and execute

**kustomize edit set image shekeriev/k8s-environ:latest=shekeriev/k8s-environ:blue**

And this one in the **green** folder

**kustomize edit set image shekeriev/k8s-environ:latest=shekeriev/k8s-environ:green**

*The changes are written in the respective* ***kustomization.yaml*** *file*

Now, we are ready

Return to the **part1/2-kustomize** folder

By now, we should have the following

**.**

**├── base**

**│   ├── appa.yaml**

**│   └── kustomization.yaml**

**└── overlays**

**├── blue**

**│   ├── custom-env.yaml**

**│   ├── custom-rs.yaml**

**│   └── kustomization.yaml**

**└── green**

**├── custom-env.yaml**

**├── custom-np.yaml**

**└── kustomization.yaml**

**4 directories, 8 files**

Check the manifests of both variants

**kustomize build overlays/blue**

**kustomize build overlays/green**

Spin them up with

**kustomize build overlays/blue | kubectl apply -f -**

**kustomize build overlays/green | kubectl apply -f -**

*Or we can go directly with*

***kubectl apply -k overlays/blue***

***kubectl apply -k overlays/green***

Check the resources we created

**kubectl get pods,svc**

Open a browser tab and navigate to http://<control-plane-ip>:30001

Open another tab and navigate to http://<control-plane-ip>:30002

Remove all traces with

**kustomize build overlays/blue | kubectl delete -f -**

**kustomize build overlays/green | kubectl delete -f -**

*Or we can go directly with*

***kubectl delete -k overlays/blue***

***kubectl delete -k overlays/green***

### Skaffold

Even though, the **Skaffold** tool is included in this section, its focus is in another direction

#### Install

For Intel compatible 64-bit Linux distributions we can install it with

**curl -Lo skaffold https://storage.googleapis.com/skaffold/releases/latest/skaffold-linux-amd64 && sudo install skaffold /usr/local/bin/**

For other options, please visit <https://skaffold.dev/docs/install/#standalone-binary>

Once installed, check it is working with

**skaffold version**

#### See it in Action #1

Without wasting more time, let's start it and see it in action

Navigate to the **part1/3-skaffold** folder and explore the files

To create the project, execute

**skaffold init**

Accept the proposed configuration by hitting **Y**

We can see that one new file was added – **skaffold.yaml**

*Please note that you should adjust the image repository (both in the* ***pod.yaml*** *and* ***skaffold.yaml*** *files) to match your situation (your repository and image tag)*

As we will be testing it against a **Kubernetes** cluster, we should have a place to publish the container image

For this sample, we will use **Docker Hub** as a registry

Make sure that you logged in

**docker login**

*Before starting the project, check if the* ***code.sh*** *file has been set as executable and if not, make it*

Now, start our project with

**skaffold run --tail**

It is working 😊

Press **Ctrl+C** to stop the log following

And then delete the pod

**kubectl delete pod skaffold**

#### See it in Action #2

Now, let's see the automatic redeployment capabilities of **skaffold**

In the first (existing) terminal session, execute this

**skaffold dev**

Then open another terminal session and open for example the code.sh file for editing

Modify the greeting text for example to **Hello Beautiful World!** and save it

Return to the first terminal session and watch

The image will rebuild and after a while the pod will change

Return to the second terminal and change something else. For example, the interval from **2** to **5** seconds and then save and close the file

Return to the first terminal session and watch

The image will rebuild and after a while the pod will change

Once done, experimenting, hit **Ctrl+C** (in the first terminal)

That is all (at least for this session). There is plenty to explore here (as with all other topics we covered so far) but I will leave this to you 😉

## Part 2: Getting Started with Helm

First, we will install the helm binary and then experiment with a few charts

### Installation

There are multiple ways to install **Helm** and perhaps the easiest one is by downloading the binary file

For any 64-bit Linux distribution this can be done in the following way

**curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash**

*The above command assumes that the appropriate permissions are in place and we trust the source*

Latest version for other OSes can be downloaded from here: <https://github.com/helm/helm/releases/latest>

Other options and further instructions can be found here: <https://v3.helm.sh/docs/intro/install/>

Once, we have it installed, we can test if it is working by executing

**helm version**

### First Charts

Before we can start working with charts, we must register one or more repositories

The easiest way to find some, is to use the **ArtifactHub** by visiting this address: <https://artifacthub.io/>

Enter **nginx** in the search box and press **Enter**

A list of **nginx** related charts will appear

The first one should be a one provided by **Bitnami**

Click on it (or visit this URL: <https://artifacthub.io/packages/helm/bitnami/nginx>)

There, in the beginning, are the instructions on how to add the repository

So, let's do it

**helm repo add bitnami https://charts.bitnami.com/bitnami**

Now, we can list all locally registered repositories (it should be just this one)

**helm repo list**

Okay, but where the information about the repository has been stored?

Let's see where with

**helm env**

Aha, here it is

Let's check what is behind the **HELM\_REPOSITORY\_CACHE** and **HELM\_REPOSITORY\_CONFIG** variables

The first one (**HELM\_REPOSITORY\_CACHE**) points to a folder, where information (the packages they provide) about the registered repositories will be stored including the charts that we use

The second one (**HELM\_REPOSITORY\_CONFIG**) points to a file which contains information about the registered repositories

#### NGINX

Now, let's install the chart that we wanted

**helm install my-nginx-1 bitnami/nginx**

We can notice two sets of information

The first one, right after our command, is a short information on what we deployed

The second one, contains detailed instructions on how to interact with what we deployed

Let's see what we have so far

**helm list**

Our only chart *(in fact a* ***release****, as it is deployed in the cluster)* is shown

We can get detailed information about the release with

**helm status my-nginx-1**

In fact, we have seen this information already

Let's use the **kubectl** to check what we have

**kubectl get pods,svc**

One pod and a service which appears to be of type **LoadBalancer** by default

Is this all? We can check with

**helm get manifest my-nginx-1**

There is one more object – a **Secret**

**kubectl get secret**

It is used to store some additional configuration for the **nginx** release

But is this really all information that we can get? No, it is not. Let's try this command

**helm get all my-nginx-1**

Wow, way more than we saw earlier. Which of the options *(****all****,* ***hooks****,* ***manifest****,* ***notes*** *or* ***values****)* to use, depends on what we are looking for

Let's try to access the default web page of the **nginx** instance

Ask for the service first

**kubectl get service my-nginx-1**

Then open a browser tab and navigate to http://<control-plane-ip:port>

Okay, it is the default page. How can we change it?

Return to the chart's page - <https://artifacthub.io/packages/helm/bitnami/nginx>

Explore the **Parameters** section

There is plenty of them, but we are interested in the **Custom NGINX application parameters** sub-section

And more specially in the **staticSiteConfigmap**

We can use it to pass a custom index page to the **nginx** chart

Let's do it. We will create a second nginx release but this time with a custom index page

First, create the configuration map with

**kubectl create configmap my-nginx-2-index --from-literal=index.html='<h1>Hello from NGINX chart :)</h1>'**

Check that it is there, and the content is as expected

**kubectl get cm**

**kubectl get cm my-nginx-2-index -o yaml**

Sure, everything is just fine

Now, install the chart but this time execute this

**helm install my-nginx-2 bitnami/nginx --set staticSiteConfigmap=my-nginx-2-index --dry-run**

This will do as stated – a dry run and show to us what will be the outcome

It seems that everything is fine, so let's re-execute the command but without the **--dry-run** option

**helm install my-nginx-2 bitnami/nginx --set staticSiteConfigmap=my-nginx-2-index**

Check what we have by now with

**helm list**

**kubectl get pods,svc**

Copy the service port and open a browser tab and navigate to http://<control-plane-ip:port>

Our custom index page is there 😊

Now, let's clean a bit

Remove the releases by executing

**helm uninstall my-nginx-1 my-nginx-2**

We can then check with

**kubectl get pods,svc,cm,secret**

Aha, most of the resources are gone, but our configuration map is not. Why? *Perhaps, because it was created separately and not as part of the chart*

Remove it

**kubectl delete cm my-nginx-2-index**

#### Apache

Let's try another one

Navigate again to the **ArtifactHub** by visiting this address: <https://artifacthub.io/>

Enter **apache** in the search box and press **Enter**

A list of **apache** related charts will appear

The first one should be a one provided by **Bitnami**

Click on it (or visit this URL: <https://artifacthub.io/packages/helm/bitnami/apache>)

We have the repository installed, thus we can proceed with the chart

First, explore the **Parameters** section on the chart's page

Now, that we are familiar with the process, let's first create a custom index.html page

**kubectl create configmap my-apache-index --from-literal=index.html='<h1>Hello from Apache chart :)</h1>'**

Next, install the chart but with this command

**helm install my-apache bitnami/apache --version=9.1.0 --set htdocsConfigMap=my-apache-index**

Check with

**helm list**

**kubectl get pods,svc,cm,secret**

Test that the web server is working as expected by visiting http://<control-plane-ip:port>

It is working 😊

Now, we suddenly realize that there is a newer version of the chart which among other improvements, provides a newer version of the **Apache** web server

Let's upgrade what we have installed (but again, first a dry run)

**helm upgrade my-apache bitnami/apache --dry-run**

Now, the actual upgrade

**helm upgrade my-apache bitnami/apache**

Check the listing

**helm list**

Yes, it shows that we are on the newer version and that the **revision** of our release is now **2**

If we check, the application should still work and showing our custom index page

What if we decide that we want to do a rollback? Can we do it? Yes, we can 😉

First, let's check the history with

**helm history my-apache**

Now, we can do a dry run of a roll back to revision 1

**helm rollback my-apache 1 --dry-run**

It appears to be successful, so let's execute it

**helm rollback my-apache 1**

We can check what is going on with

**helm history my-apache**

**kubectl get pods,svc,cm,secret**

After a while, the new pod will be the one that will serve client requests

We can check that the web server is working as expected by visiting http://<control-plane-ip:port>

Let's clean again

**helm uninstall my-apache**

**kubectl delete cm my-apache-index**

### A More Complex Chart

Why not we try something different? Perhaps, a multi-pod chart? 😉

Let's search for **wordpress** for example, but this time on the command line

We can search the hub

**helm search hub wordpress**

Or the registered repositories *(currently only one)*

**helm search repo wordpress**

Okay, let's ask for some information with

**helm show chart bitnami/wordpress**

And then with

**helm show readme bitnami/wordpress**

Wow, plenty of information. A better version can be found here: <https://artifacthub.io/packages/helm/bitnami/wordpress>

Try to install it with

**helm install my-wordpress bitnami/wordpress**

So, the process doesn't differ if the chart brings just a pod and service or a bigger set

Let's ask for the status

**helm list**

**kubectl get pods,svc**

Okay, the chart is deployed, but the pods are in **Pending** status

If we do some research, we will see that there is a requirement for persistent volumes and because we do not have them, the persistent volume claims are failing and thus chart's pods

Uninstall the chart

**helm uninstall my-wordpress**

Check that there aren't any persistent volume claims left

**kubectl get pvc**

And if any, delete them

Now we have two options – to install the chart without persistence or provide the required persistent volumes

The first option would look like this

**helm install my-wordpress bitnami/wordpress --set service.type=NodePort \**

**--set persistence.enabled=false \**

**--set mariadb.primary.persistence.enabled=false**

The second option, would require some additional work (let's skip it, as our focus now is not on persistence)

And check the related objects

**kubectl get pods,svc,pvc,pv -o wide**

After a while the pods will be in running state

Should we want, we can watch the logs of the **wordpress** pod

**kubectl logs pod/my-wordpress-<id> --follow**

Once, we are sure that everything is up and running, we can use the information provided when we deployed the chart to connect and start using the app

Or if we missed it, we can always ask for it with

**helm status my-wordpress**

So, to get the connectivity data, we can execute

**export NODE\_PORT=$(kubectl get --namespace default -o jsonpath="{.spec.ports[0].nodePort}" services my-wordpress)**

**export NODE\_IP=$(kubectl get nodes --namespace default -o jsonpath="{.items[0].status.addresses[0].address}")**

**echo "WordPress URL: http://$NODE\_IP:$NODE\_PORT/"**

**echo "WordPress Admin URL: http://$NODE\_IP:$NODE\_PORT/admin"**

And for the credentials

**echo Username: user**

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

Then, we can open a browser tab and navigate to either of the URLs we got and use the user and password we extracted

Happy blogging 😉

Once done, we should clean

**helm uninstall my-wordpress**

## Part 3: Creating Simple Charts

Let's get familiar, even on an entry level, with the process of creating our own charts

### Getting Started

First, we must prepare our playground

Navigate to a folder of your choice and execute

**helm create mychart**

Then we can check the hierarchy that has been created for us

**tree mychart**

Wow, plenty of files

Even though a good start, there are too many techniques, both **Kubernetes** and **Helm** related, applied here

Browse the folders and explore the files

It is better to start from a clean slate

Create another set of folders with

**mkdir -p webchart/templates**

**touch webchart/{Chart.yaml,values.yaml} webchart/templates/{cm.yaml,pod.yaml,svc.yaml}**

Now, check our hierarchy

**tree webchart**

Before we start filling the files, let's set what we are aiming for

We want to build a chart that will spin a pod with single **apache**-based container it will mount a config map with a custom index page and will expose the web server via a service

So, three objects in total, hence the number of files in the **webchart/templates/** folder

Let's start with the **webchart/Chart.yaml** file. We should put there the following

apiVersion: v2

name: webchart

description: A simple Apache-based Helm chart for Kubernetes

# Chart type. A chart can be either an 'application' or a 'library' chart

type: application

# Version of the chart

version: 0.1.0

# Version of the application. In our case - Apache 2.4.51

appVersion: "2.4.51"

More on versioning can be read here: <https://semver.org/>

We will leave the **webchart/values.yaml** file empty for now

Let's enter the following in the **webchart/templates/cm.yaml** file

apiVersion: v1

kind: ConfigMap

metadata:

  name: webchart-cm

data:

  index.html: <h1>Hello from Apache :)</h1>

Nothing special here. Next stop is the **webchart/templates/pod.yaml** file

apiVersion: v1

kind: Pod

metadata:

  name: webchart-pod

  labels:

    app: webchart

spec:

  containers:

  - name: main

    image: httpd:2.4.51

    ports:

    - containerPort: 80

    volumeMounts:

    - name: html

      mountPath: /usr/local/apache2/htdocs

  volumes:

  - name: html

    configMap:

      name: webchart-cm

Last, but not least is the **webchart/templates/svc.yaml** file

apiVersion: v1

kind: Service

metadata:

  name: webchart-svc

spec:

  type: NodePort

  ports:

  - port: 80

    protocol: TCP

  selector:

    app: webchart

Now, let's test our chart. First, we will ask helm for information

**helm show chart webchart**

Even if we change chart to all, we won't see much information

So, let's install it with

**helm install test1 webchart**

Then we can check with

**helm list**

And then with

**helm get manifest test1**

And finally, with

**kubectl get cm,pods,svc**

Open a browser tab and navigate to our application http://<control-plane-ip>:<port>

It is working 😊

Let's try to install another release

**helm install test2 webchart**

Ha, it doesn't work … ☹ Why? *Perhaps, because of the hardcoded names of the resources …*

Let's fix this but first, uninstall the release

**helm uninstall test1**

As a minimum, we must parametrize the names of the resources

But what to put there? The release name is one of the best candidates. So, let's use it

Open the **webchart/templates/cm.yaml** file and modify it to look like this

apiVersion: v1

kind: ConfigMap

metadata:

  name: {{ .Release.Name }}-cm

data:

  index.html: <h1>Hello from Apache :)</h1>

This is one of the built-in objects that we mentioned in the slides

Next, open the **webchart/templates/pod.yaml** file and modify it to look like this

apiVersion: v1

kind: Pod

metadata:

  name: {{ .Release.Name }}-pod

  labels:

    app: {{ .Release.Name }}

spec:

  containers:

  - name: main

    image: httpd:2.4.51

    ports:

    - containerPort: 80

    volumeMounts:

    - name: html

      mountPath: /usr/local/apache2/htdocs

  volumes:

  - name: html

    configMap:

      name: {{ .Release.Name }}-cm

Please note, that we must change not only the pod's name but also the label key-value pair and the config map reference

Finally, open the **webchart/templates/svc.yaml** file and modify it to

apiVersion: v1

kind: Service

metadata:

  name: {{ .Release.Name }}-svc

spec:

  type: NodePort

  ports:

  - port: 80

    protocol: TCP

  selector:

    app: {{ .Release.Name }}

We are done. Let's try to install one release

**helm install test1 webchart**

And then check how the resources are being created

**helm list**

**kubectl get cm,pods,svc**

Now, try to install a new release

**helm install test2 webchart**

And then again, check how the resources are being created

**helm list**

**kubectl get cm,pods,svc**

Everything is working as expected 😊

Perhaps, we can do more. For example, use a deployment instead of a bare pod, and expose some values for the user to configure

We will do this in the next section

Now, let's clean the releases

**helm uninstall test1 test2**

Should we want, we can package our chart and then publish it to a repository somewhere

Even though, this is not within the scope of the current module, let's at least package it

**helm package webchart**

That is it

### Chart From Existing Manifest

Now, let's do the things in another way. Let's start from an existing set of manifests and turn it to a chart

Create a new folder **appchart** and a sub-folder **templates**

**mkdir -p appchart/templates**

Copy the files from the **part3/appchart-1-start** folder to the **appchart/templates** folder

**cp -v part3/appchart-1-start/\*.yaml appchart/templates/**

Now, create the **appchart/Chart.yaml** file with the following content

apiVersion: v2

name: appchart

description: A simple Helm chart for Kubernetes based on an existing application

# Chart type. A chart can be either an 'application' or a 'library' chart

type: application

# Version of the chart

version: 0.1.0

# Version of the application. In our case this can be a custom version as it is our application

appVersion: "1.0.0"

Create an empty, for now, **appchart/values.yaml file**

**touch appchart/values.yaml**

Open the **appchart/templates/deployment.yaml** file and modify it to match this

apiVersion: apps/v1

kind: Deployment

metadata:

  name: {{ .Release.Name }}-deployment

spec:

  replicas: {{ .Values.replicasCount }}

  selector:

    matchLabels:

      app: {{ .Release.Name }}

  template:

    metadata:

      labels:

        app: {{ .Release.Name }}

    spec:

      containers:

      - name: main

        image: shekeriev/k8s-environ:latest

        env:

        - name: APPROACH

          value: {{ .Values.approachVar }}

        - name: FOCUSON

          value: {{ .Values.focusOnVar }}

Here, we extend the level of parametrization by utilizing the **appchart/variables.yaml** file (we will fill it with content a little bit later)

Next, edit the **appchart/templates/service.yaml** file to match this

apiVersion: v1

kind: Service

metadata:

  name: {{ .Release.Name }}-service

spec:

  type: NodePort

  ports:

  - port: 80

    nodePort: {{ .Values.nodePort }}

    protocol: TCP

  selector:

    app: {{ .Release.Name }}

It is time to put some content in the **appchart/values.yaml** file

replicasCount: 1

approachVar: "Helm Charts! :)"

focusOnVar: "APPROACH"

nodePort: 30001

Now, let's test our newly created chart

**helm show all appchart**

And install it

**helm install app1 appchart**

Check how things went

**helm list**

**kubectl get deployments,pods,svc**

Open a browser tab and navigate to http://<control-plane-ip>:30001

Super! It is working 😊

Now, let's try to install another release

**helm install app2 appchart**

Hm, we got an error. Why? *Perhaps, because we did not change any of the values …*

Let's check if there are any resources

**kubectl get deployments,pods,svc**

Yes, there are. Let's uninstall it

**helm uninstall app2**

And try to install it again but this time with this command

**helm install app2 appchart --set nodePort=30002 --set replicasCount=3**

Now, check

**helm list**

**kubectl get deployments,pods,svc**

It is working 😊

If we open a browser tab, we can confirm that not only the resources are there, but also the application is working

There are further improvements that could be made here. For example, we may put some logic to have a random value for the node port if one is not provided. *You can try to tackle this in your spare time 😉*

Let's clean by uninstalling the releases

**helm uninstall app1 app2**