Introduction to Helm and the problems that it solves

In the previous section, I gave a very quick introduction about Helm.

Now, inside this lecture, let us try to understand more details about Helm.

**If someone asks you what is Helm, you can simply reply saying that Helm is a package manager for Kubernetes.**

**The main objective of Helm is to help developers and DevOps team members to manage Kubernetes projects and deployments by offering a more efficient approach in handling the Kubernetes manifest files.**

Regardless of how many microservices you have inside your microservice network, Helm is going to make your life easy without helm like we discussed previously, we need to maintain lot many Kubernetes manifest files. Without helm like we discussed previously, we need to maintain all the Kubernetes manifest files like for deployment, service, config map, for each microservice that you are going to deploy inside your microservice.

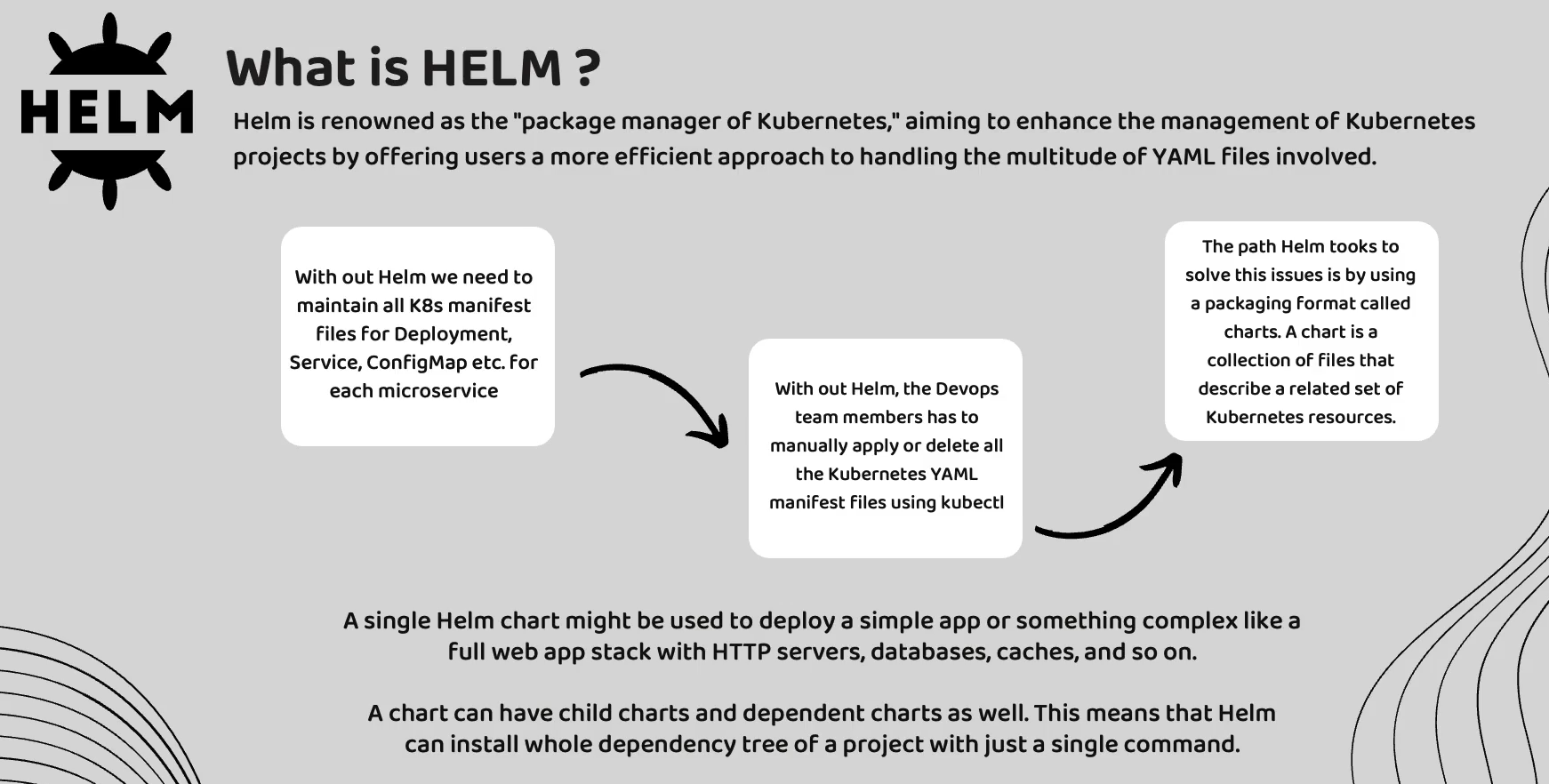
And apart from maintaining these Kubernetes manifest files, your DevOps team members also they should manually apply or delete these Kubernetes manifest files with the help of **kubectl** commands.

But like I said, if we use Helm, we can overcome these challenges.

And here you may have question like how Helm is going to help us to overcome these challenges.

For the same helm is going to follow a **packaging format called Charts**.

Inside Helm, a chart is a collection of files that describes a related set of Kubernetes resources.



Suppose inside your microservice network, if you have 50 microservices, you can club all these 50

microservices related manifest files into a single component called Chart inside Helm.

So, using the same single chart, we can deploy a simple application or any kind of complex application which can include Http servers, REST APIs, databases, supporting components like cache.

So regardless of how complex your microservice network is, you can always try to create a chart specific to your organization.

And with that, the maintenance of your Kubernetes deployments is going to be super easy.

And one more beauty of this helm charts is, a chart can have a **child charts** or **dependent charts** as well, just like in Java, how we can have parent class or child class or dependent classes very similarly

inside helm charts also, we can maintain the dependent charts and child charts in case if you have dependency on other third-party components, you can try to add the helm charts of those components into your organization charts.

This way you can install whole dependency tree of your project, including your own microservices and dependent components with just a single command.

And like I said, Helm is a package manager for Kubernetes.

And here you may have a question.

**What is a package manager?**

**Package manager** is a component that can help you in installing or uninstalling or upgrading your software packages. Just like how we have helm for Kubernetes, very similarly, we also have other famous package managers inside the industry.

*For example,* ***Pip*** *is a package manager that can help you to install any Python packages.*

*And very similarly for JavaScript, we have a package manager with the name* ***NPM****.*

*Using this NPM package manager, we can install any kind of JavaScript libraries like Angular, React.*

*All these libraries we can easily install and set up inside our local system with the help of this NPM package manager.*

Very similarly, ***helm*** **is the best way to find, share and use software’s built for Kubernetes**.

I hope this is clear.

Now let me try to explain you very quickly on how Helm is going to solve all the challenges that we

have discussed. Without Helm like we discussed, we need to maintain separate Kubernetes manifest files for all the microservices inside a project.

For example, think like I have accounts, loans, and cards, microservice. If I want to access them,

either using external traffic or with the help of internal traffic, I should expose them with the help

of Service object inside Kubernetes.

For the same, I need to create three different Kubernetes manifest files, one for account-service,

and the second one is for the loan-service and the third one for the card-service.

So, if you try to closely observe these manifest files, the skeleton of these files is almost same or

static except a few dynamic values, like the very first element **apiVersion:** which is v1 is going

to be static for all the files.

The same applies for **kind:** object post that will be having **metadata** **name** inside the metadata name,

we are going to have a dynamic value which is specific to this microservice.

Post that and a **spec** and **selector**.

The app name is going to be the dynamic value and the type also

is going to be a dynamic value based upon our requirements.

We can use LoadBalancer or ClusterIP, but coming to the **protocol**, TCP is going to be a static value

and the **port** and **targetPort** will have dynamic values which are specific to each microservice.

So apart from these few dynamic values, the whole skeleton or template of the Kubernetes service manifest file is going to be the same.

This is where Helm is going to come into picture and it will try to make our life easy.

Now, let me quickly show you how Helm is going to approach this problem.

So, whenever we are using Helm, first, we are going to create a single **template.yaml** file for the Service object regardless of how many microservices you have inside your cluster, you are always going to have a single **template.yaml** file like you can see here.



This template yaml files will have static values along with the static syntax structure.

If you closely observe this **helm service template file** at some places it is trying to accept the dynamic values.

For example, under the metadata name it is going to accept the dynamic value with the help of name: {{ .Values.serviceName }}

type: {{ .Values.service.type }}

targetPort: {{ .Values.service.targetPort }}

So, whatever you mention inside this double curly brace is going to represent a dynamic value which

we are going to **inject at runtime whenever we are trying to deploy our microservices into Kubernetes cluster.**

For example, think like we have this helm service template file. As a next step, what I can do is, I can try to provide a **values.yaml** file which is specific to accounts microservice.

So inside this values.yaml I'm going to maintain all the key and values you see inside my values.yaml



The very first property that I am trying to create here is **deploymentlabel: accounts**

The same value that we have defined inside the values.yaml is going to be applied to the service

template file at runtime and behind the scenes helm is going to automatically create the account microservice specific service manifest file.

Very similarly, you just have to provide the values.yaml for other microservices.

With that, you will always have a single template file which is going to be maintained by the helm.

But as a developer or as a DevOps team member, we should provide the values.yaml for each microservice.

And behind the scenes, using these values that we have defined, the helm is going to generate the Kubernetes manifest files at runtime for all your microservices.

**On top of that, we do not have to execute these Kubernetes manifest files manually.**

Whenever we are using Helm, we are going to achieve all the deployment of our microservice with a single command.

So, by following this approach, Helm is going to solve these problems.

**What are the problems that helm try to solve?**

**1st problem solved**

The very first advantage, I would say as Helm is going to support packaging of your Kubernetes manifest files into a single helm chart, the same helm chart can be distributed into a public repository or a private repository you can share with others.

So, it is up to you how you want to leverage that helm chart.

This is very similar to our Java code.

We can store that in a centralized location if needed

we can share our code with others.

Very similarly, we are going to maintain the helm charts.

That is why we are calling Helm as a package manager for Kubernetes.

**2nd problem solved**

And the next problem that Helm solves is it is going to make developers or DevOps team members life easy because it is going to support easier installation.

With the help of Helm, you can always deploy or upgrade or rollback or uninstall your entire microservice application into your Kubernetes cluster with a single command.

No need to run any manual commands with the help of Kubectl, and at last Helm is going to support,

release or version management as well, which means whenever needed, you can roll back your entire Kubernetes cluster to the previous working state with a single command.

Whereas with Kubernetes manifest files like we discussed previously, we can only roll back a particular microservice.

What if you want to roll back your entire Kubernetes cluster. In such scenarios, helm is going to help us.

Installing Helm

Install Chocolatey

<https://chocolatey.org/install>

<https://helm.sh/docs/intro/install/>

Installing a sample Helm Chart

Now, inside this lecture, I am going to show you the power of hell by taking a sample chart available

from the official documentation.

So, before we try to explore the official documentation of hell on how to install a chart of hell, first,

we need to make sure the local Kubernetes cluster is up and running.



We can confirm by going to the Docker dashboard inside the Docker dashboard, you should be able to see a message which is Kubernetes is running.

So, this confirms the cluster is running.

Very similarly you can also run any of the Kubectl command, for example, I can try to run a command which

**> kubectl get services**

So, this will give me the list of services inside my local Kubernetes cluster.

As of now, we have only one service which is related to Kubernetes itself.



So, this confirms our local Kubernetes cluster is working fine.

As a next step, I am going to run a helm command, which

**> helm ls**

So, this is going to list all the releases or all the installations that we have done into the Kubernetes

cluster with the help of Helm.

So, if I try to run as of now, you can see there is no helm installations because as of now we have

not installed any chart into the Kubernetes cluster.

**How my helm can connect to the Kubernetes cluster**?

Because without connecting to the Kubernetes cluster, my helm cannot show this output.

So, the answer to this question is, Helm is going to look for Kubernetes cluster connection details inside your local system.

So, whenever you try to connect to a Kubernetes cluster inside your local system, it is going to make

an entry inside your system.

For example, inside my scenario, I can go to folder

**C:\Users\niles\.kube**

If I try to open this, there is a config file, so if I try to open this file, you will be able to see

all the connection details that my kubectl right now is using to connect to my Kubernetes cluster.

So, you can see right now it is connected to a Docker desktop cluster.

So, the same connection details Helm also is going to leverage whenever it wants to interact with a Kubernetes cluster.

I hope this is clear.

If you are using Windows Operating system, you will be able to find **C:\Users\niles\.kube**

Now let us go and explore on how to setup a sample helm chart inside our local system.

So that will give some good exposure to the helm.

And what are the advantages of Helm?

So, for the same inside this website where we explored about the installation of helm into the local

system, just under this installing helm, we also have one more option which is using Helm.

<https://helm.sh/docs/intro/using_helm/>

So let me click on that post that here there is some good introduction on how to use helm.

<https://helm.sh/docs/intro/using_helm/#helm-search-finding-charts>

You can read if you are interested, but I want to directly go to the section where we have

**’helm search’** command.

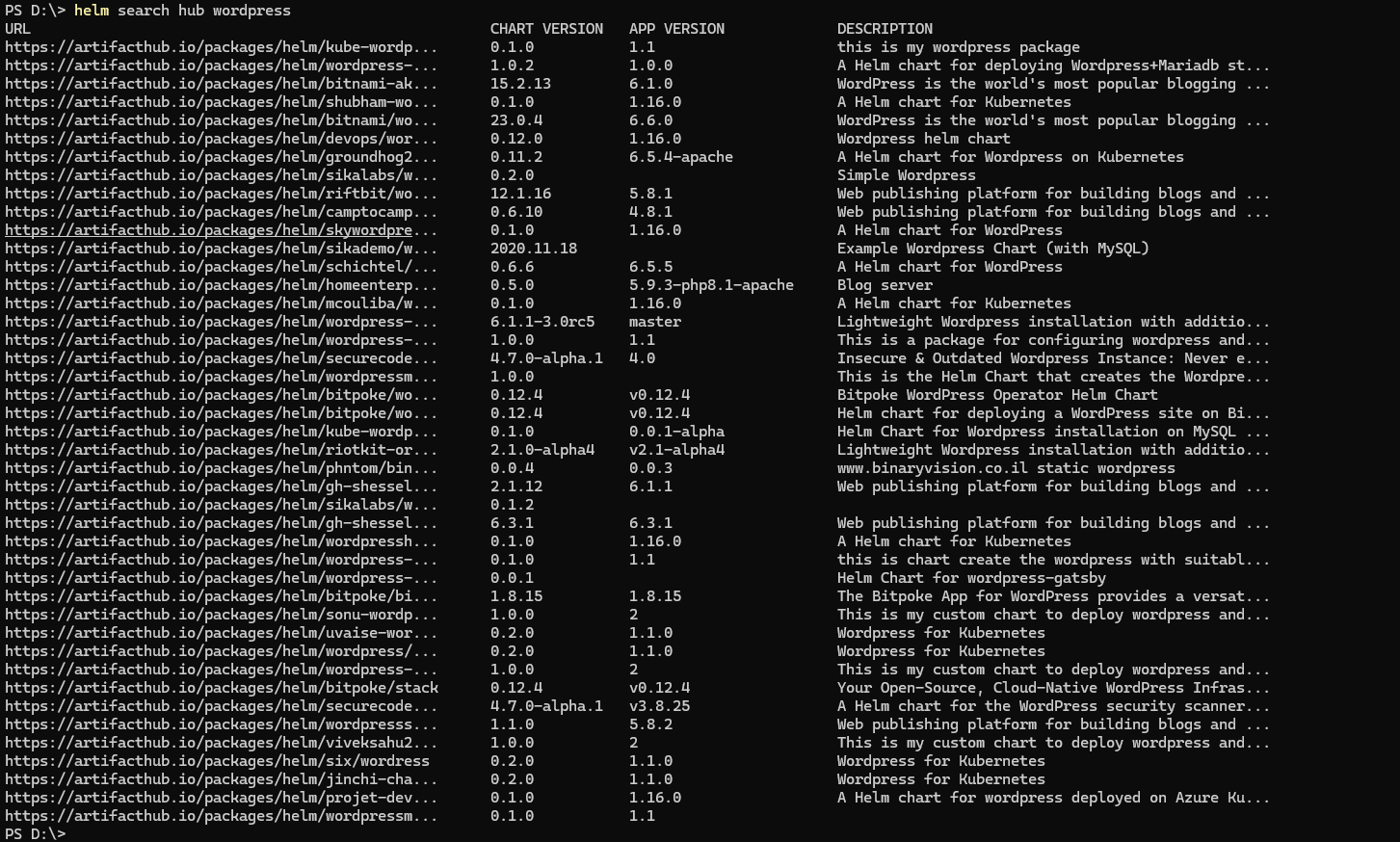
Helm comes with a powerful search command.

Like I said before, there is a good amount of community for Helm where a lot many companies, organizations, and open-source developers, they build a lot of helm charts which you can search with the help of helm search.

For example, here we have a command which is   
**> helm search hub wordpress**

Think like you want to deploy a wordpress website into your Kubernetes cluster. For the same,

first, we need to search if there are any wordpress **related charts available inside the public repositories**.



As soon as I enter, you will be able to see all the repositories where we have a chart with the name wordpress.

So, you can see a lot many places we are able to see wordpress.

So that is why we can see all of them in the output.

Here also, you can see there is a statement saying that

*“The above searches for all wordpress charts on Artifact Hub”*   
So, there are many repositories output that we receive.

So, we have received many repository details based upon our helm search command.

So, we need to choose one of the repositories to install the wordpress helm chart.

For the same, let us go to the website here.

<https://helm.sh/docs/intro/using_helm/#helm-install-installing-a-package>

If you can scroll down, there is a helm install section here they have mentioned on how to install

the WordPress chart available under the Bitnami repo.

> helm install happy-panda bitnami/wordpress

**How to add bitnami repo?**

Like I said, Bitnami is a famous repository which maintains production ready helm charts. So, we can

use the same Bitnami repository to install this WordPress chart.

But before we try to install this command, we need to make sure we have added the Bitnami repo details inside our local system.

So how to add that for the same you can take this bitnami and search inside the Google saying that bitnami helm chart installation.

<https://docs.bitnami.com/kubernetes/faq/get-started/install-configure-helm/>

So, if you try to search with this, you will get the results inside the Google search engine.

And here you can click on this very first link, which is install the chart. So here you can see the very first command they have given is how to add the Bitnami repo to the local helm.

Here I am trying to run the command and this will add the bitnami repo into our local system.

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



Now we can safely run this helm install command.

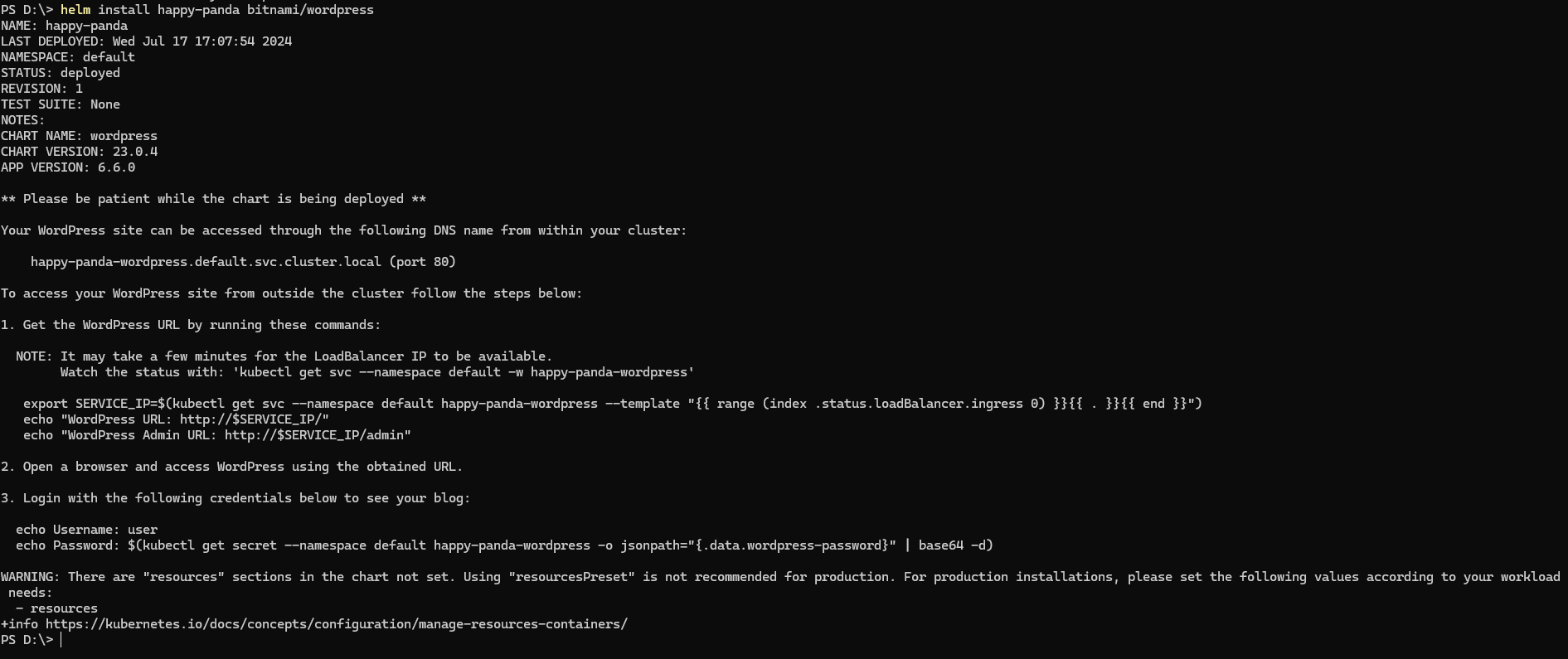
**> helm install happy-panda bitnami/wordpress**

So, the command is helm install and what is the **installation name or the release name** that you want to give? So here they have given the name as happy-panda. The name can be anything and post that bitnami indicates **what is the repository and this repo** which is **bitnami**.

What is the **helm chart**? The helm chart name is wordpress.

So, if I try to run this command, my helm is going to install the chart available inside the Bitnami

repo.



So here if you try to read the very first point, they highlighted that you can get the WordPress URL

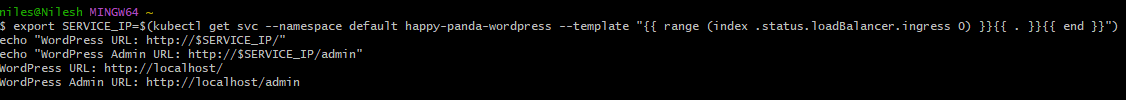
by running these commands and these are the commands.

But before that we have a note saying that it may take few minutes for the load balancer IP to be available. So, we need to be patient and we need to give 1 or 2 minutes for this to get complete.

So, to understand what is the URL of WordPress, we can try to copy this entire command and execute the same inside our terminal.

To execute this command, in Windows 11, you have a few options, including using the Windows Subsystem for Linux (WSL), Git Bash, or a similar Unix-like environment. Here is how you can do it using WSL and Git Bash:

**> wsl --install**

****

**export SERVICE\_IP=$(kubectl get svc --namespace default happy-panda-wordpress --template "{{ range (index .status.loadBalancer.ingress 0) }}{{ . }}{{ end }}")**

**echo "WordPress URL: http://$SERVICE\_IP/"**

**echo "WordPress Admin URL:** [**http://$SERVICE\_IP/admin**](http://$SERVICE_IP/admin)**"**

And this will give you an output saying that the WordPress URL is http localhost,

whereas if you want to log in as an admin, you need to access the URL, which is http localhost/admin

So let me go to the browser and try to access the same.

First, I will try to access the normal URL.

So here I am trying to access and you can see I got a sample WordPress web page.



So, what is WordPress famous for? To build the blog website. So that is why we can see some sample blog website. If you are not able to access it, please wait for 1 to 2 minutes.

You can also confirm the deployment status inside the dashboard.

\eazybank-kubernetes-manifests\kubernetes-dashboard> **kubectl apply -f .\dashboard-adminuser.yaml**

serviceaccount/admin-user created

\eazybank-kubernetes-manifests\kubernetes-dashboard> **kubectl apply -f .\dashboard-rolebinding.yaml**

clusterrolebinding.rbac.authorization.k8s.io/admin-user unchanged

\eazybank-kubernetes-manifests\kubernetes-dashboard> **kubectl -n kubectl -n kubernetes-dashboard create token admin-user**

\eazybank-kubernetes-manifests\kubernetes-dashboard> **kubectl -n kubernetes-dashboard create token admin-user**

eyJhbGciOiJSUzI1NiIsImtpZCI6IlNQOHJNUmdsY1dhbXp3UUt1ZDB6cE9MTWRYd0xhNmltLWVrVjhSTkVrZ2cifQ.eyJhdWQiOlsiaHR0cHM6Ly9rdWJlcm5ldGVzLmRlZmF1bHQuc3ZjLmNsdXN0ZXIubG9jYWwiXSwiZXhwIjoxNzIxMjIyOTQ3LCJpYXQiOjE3MjEyMTkzNDcsImlzcyI6Imh0dHBzOi8va3ViZXJuZXRlcy5kZWZhdWx0LnN2Yy5jbHVzdGVyLmxvY2FsIiwia3ViZXJuZXRlcy5pbyI6eyJuYW1lc3BhY2UiOiJrdWJlcm5ldGVzLWRhc2hib2FyZCIsInNlcnZpY2VhY2NvdW50Ijp7Im5hbWUiOiJhZG1pbi11c2VyIiwidWlkIjoiZmQ2ZGYzYzMtZjY5OC00OWQ0LThmM2EtNzMyMDkyYzM2NDAzIn19LCJuYmYiOjE3MjEyMTkzNDcsInN1YiI6InN5c3RlbTpzZXJ2aWNlYWNjb3VudDprdWJlcm5ldGVzLWRhc2hib2FyZDphZG1pbi11c2VyIn0.nXbLmaD\_OfG7OtGLNI3NoYpS7XY9Yx0YPdWSdEo\_NtRNp\_eql5rsodNVt-NHrEJrZY-wTOygVn4BjizPQBk7PcBb0sMDtR9erRcR35sKvHycxXjNlT\_STxikk3mAul0cqNkVGBKoBsI1eNszzPCTs2Xiv031OebPw3K3Cutwno2JjBalERilJ5RD0tcgDFG\_rBjN9aiNoNEr7zDMJiW23CXRG\_QbUwkQ3wj3pEqdSZO1D5on6zFbftfLWzEZmhzLZwPXNi\_lNcRDoyOUjEWoA8YzuzbcBIigTyBs7\_zygODrkZFCsgsW6sTIjXa0abaPPfFXJiWvjEisVqbWlIESpw

\eazybank-kubernetes-manifests\kubernetes-dashboard> **kubectl apply -f .\long-lived-secret-dashboard-token.yaml**

secret/admin-user created

So as of now you can see whatever helm chart that we have installed.

It did a lot of work behind the scenes.

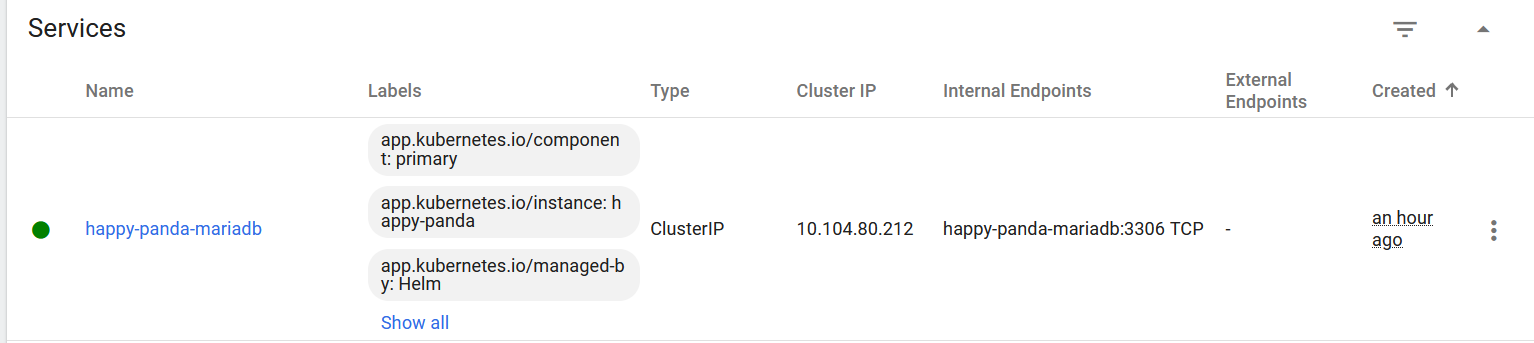
First, you can check and deployments.

There is 1 deployment happened and similarly, under the pods you will be able to see 2 pods.

One is related to the MariaDB, which my blog website is going to use behind the scenes.

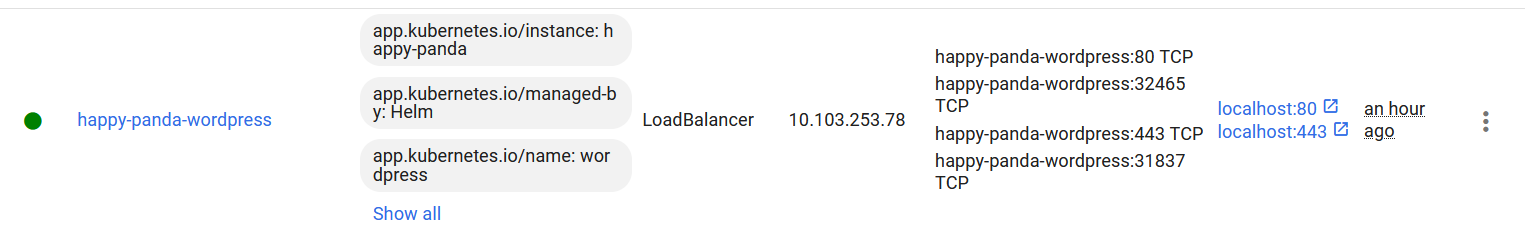
And the second part is related to the WordPress website itself and we will be having replica set details and similarly, you will be able to see the service details.

So, my MariaDB is exposed as cluster IP because I do not have any requirement to access from outside of the cluster. My WordPress website is going to access it internally within the cluster.



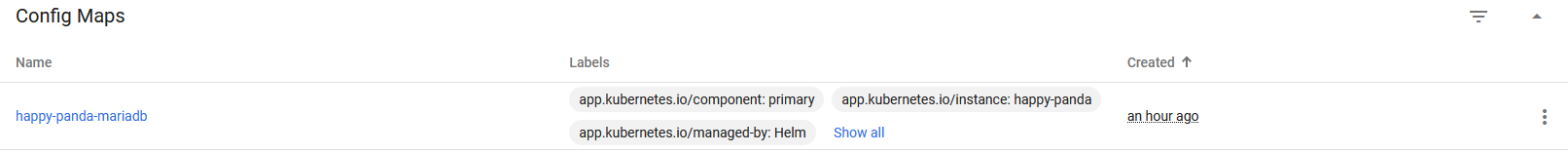
Whereas the WordPress website is deployed as LoadBalancer service type because we need to access it from outside of the cluster.

That is why we have the LoadBalancer service type here.



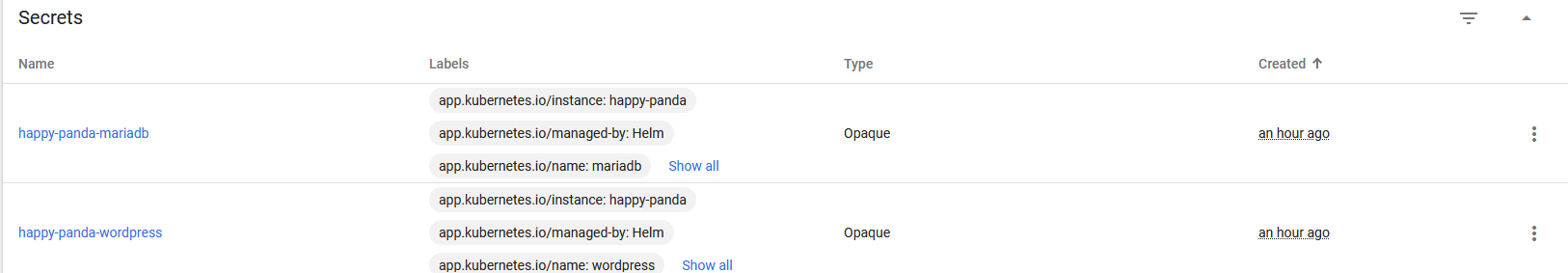
Very similarly, you can go and check what are the config maps created.

So here this is a config map which has all these environment property details



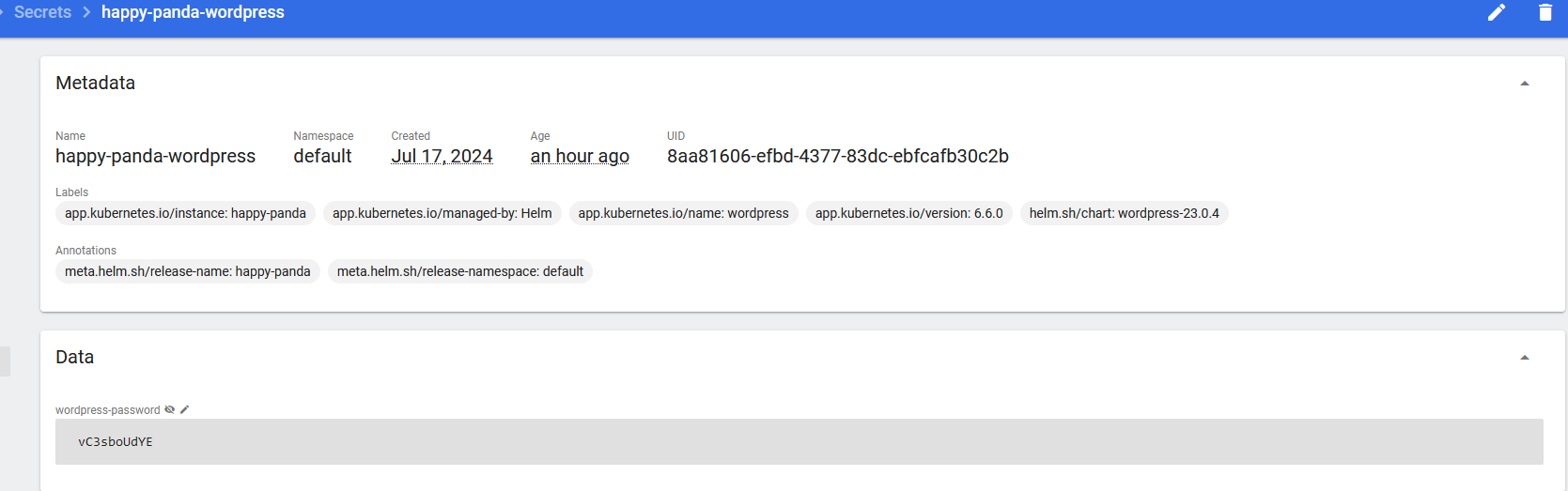
and similarly you can

go to secrets and here there are two secrets one with the name happy-panda-wordpress and the other one related to the database.



So, if I try to click on this WordPress secret, so here we have a password that we can use to access

the website as an admin.

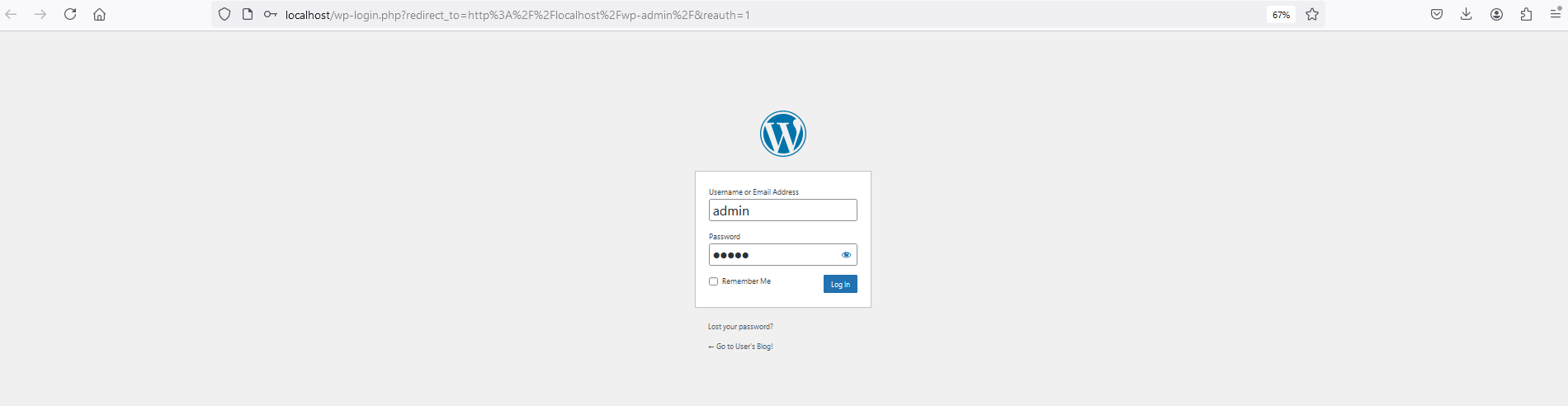


So, this way behind the scenes, my helm chart did a lot of job for me to set up this website with production ready standards.

So now let me try to access as an admin for the same,

I just need to access path http://localhost/admin,

but it is going to ask me the username and password, password we already saw previously.



This is going to be the password.

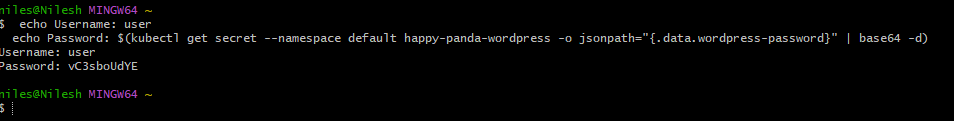
We can also try to get the username and password from the instructions that we have received on the terminal. So, inside the terminal you can see under the third point they have highlighted how to get the username and password so you can try to take this command and run the same inside your terminal.

And this will give what is the username.

Username is user and this is the password.

$ echo Username: user

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



So let me copy the password.

Here the username is going to be user and password

I have pasted and post that I am going to click on this login.

Now you can see I am able to access my WordPress website as an admin.

Here I can create new blogs or new articles as an admin of this WordPress website.

So, the point that I want you to highlight here is did you see how easily I can set up all these

WordPress websites very easily with the help of Helm chart, I just ran a single command which

**> helm install command**

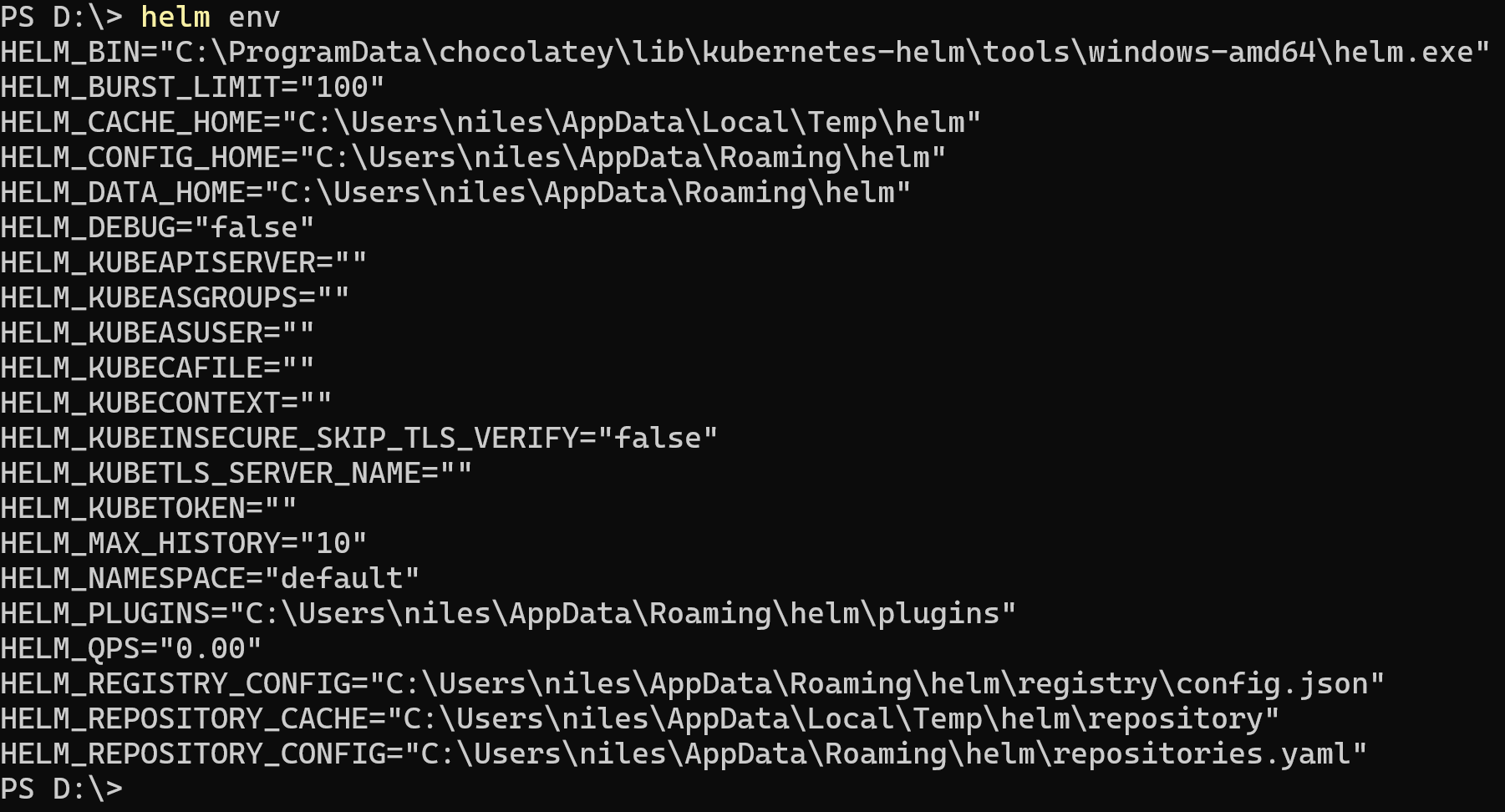
and behind the scenes it ran so many Kubernetes manifest files.

So, do you want to see what are all the files that it has executed?

So, for the same we need to understand where the helm chart inside your local system is saved.

So, to understand the same, we need to run the command which

**> helm env**

****

So, this will give you output about your health.

Here I just wanted to highlight you that all our helm charts are going to be saved under these



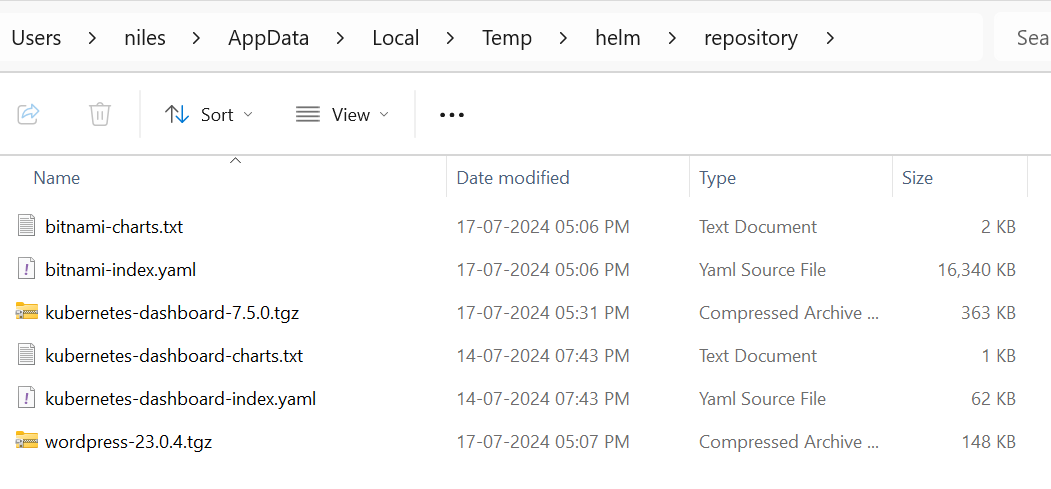
So, let me copy this path.

I will try to access the same.

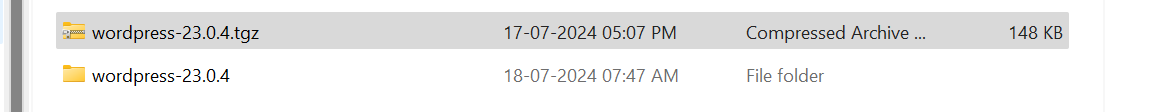
So, inside this helm folder we have another folder

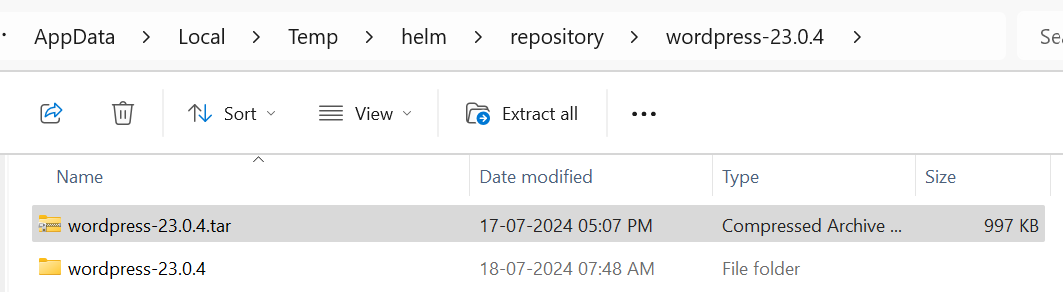
C:\Users\niles\AppData\Local\Temp\helm\**repository**

And here we have a helm chart with the name wordpress.



So, let us try to extract this compressed file.

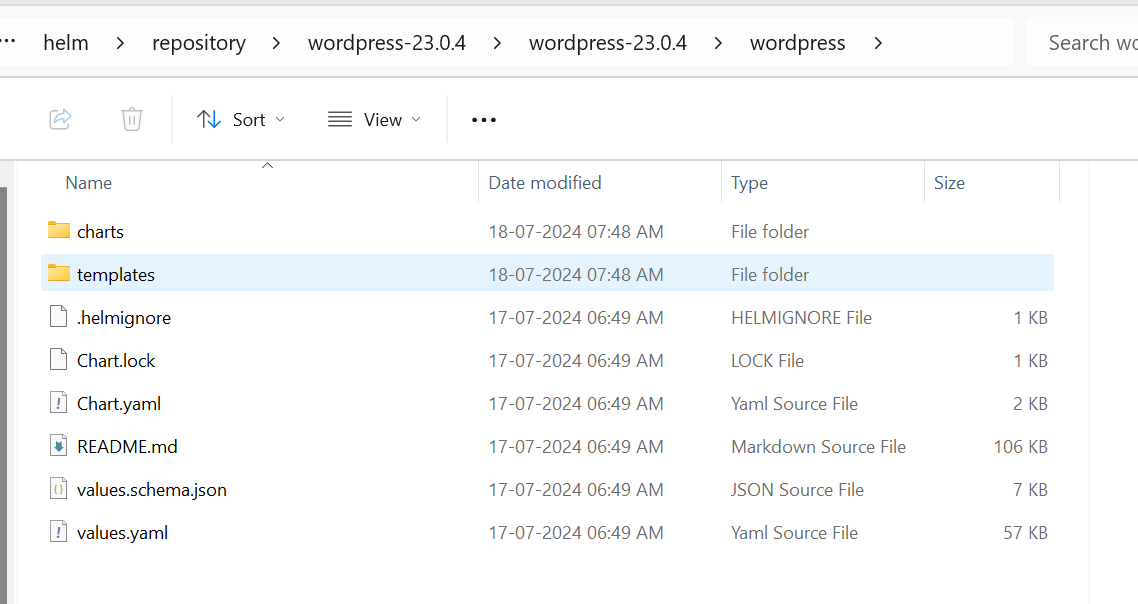




So, inside my local system I extracted this by double clicking on this file.

So, if you try to open this WordPress, you will be able to see the list of files available inside the

helm chart with the name WordPress.



So, let us try to understand more details about the structure of helm charts in the next lecture.

In the same process, I am going to show you all the Kubernetes manifest files that my helm ran behind the scenes to set up the WordPress website.

Understanding Helm Chart structure

Inside this lecture, let us try to understand what is the helm chart structure.

Any time you try to create your own helm chart or you try to use the third-party helm chart, it is

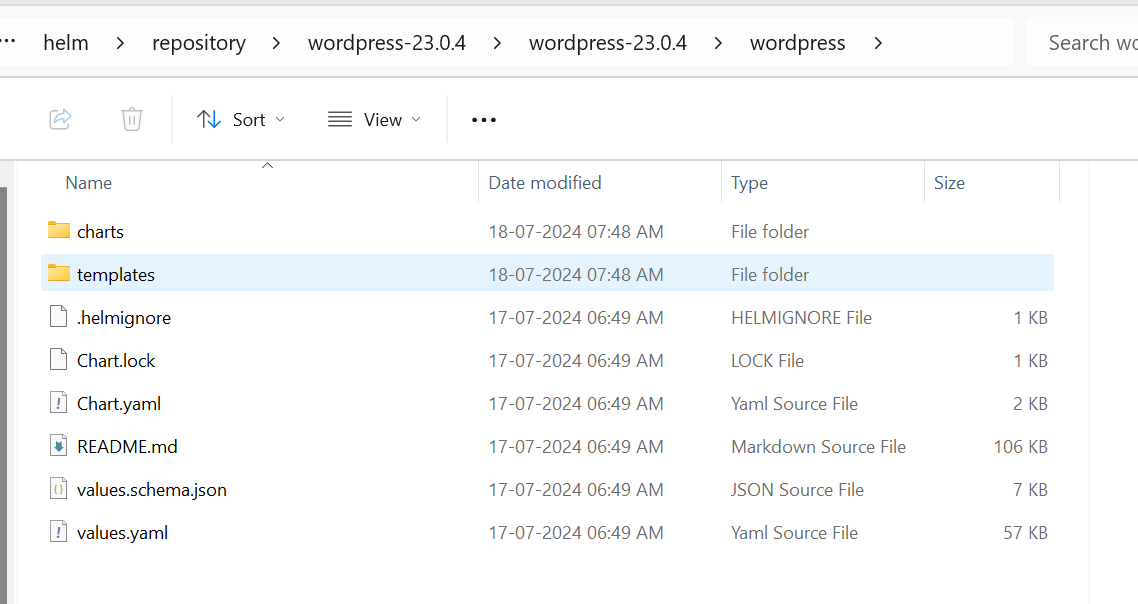
going to follow the predefined structure.

So, let us try to understand the same.

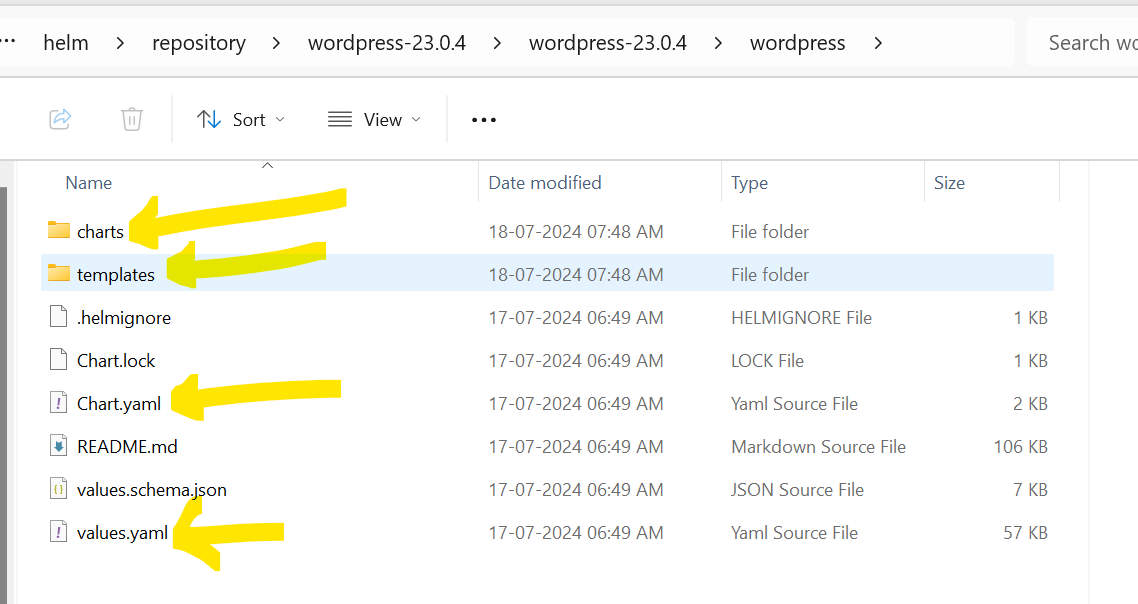
Like you can see here on the left-hand side.

First, we will be having the folder with the name of the helm chart.

In the previous lecture we installed the chart with the name wordpress.



Using the same name, a parent folder will be created inside this parent folder wordpress, we are going to have two Yaml files and two different sub folders.



So, let us try to understand about each of them.

The very first Yaml file we are going to have inside the wordpress folder is **Chart.yaml**.

So, this **Chart.yaml** file is going to have meta information about the helm chart.

Like what is the chart version? So all such kind of meta information about the chart itself is going to be maintained inside this **Chart.yaml**.

The next yaml file is **values.yaml**. Inside the **values.yaml**, we need to maintain all the dynamic values that we want to inject at runtime into the template files.

Next, we have two subfolders.

The very first sub folder is **charts**.

So, inside this **charts** folder we will have other helm charts which my current chart is dependent on.

Next, we also **templates** sub folder.

We are going to maintain all the required Kubernetes manifest template Yaml files so into this template Yaml files only at runtime my helm chart is going to inject all the dynamic values that we have mentioned inside the **values.yaml**.



So, let us try to validate the same helm chart structure by exploring the wordpress helm chart that we have created in the previous lecture.

So here, like you can see the parent folder name is wordpress because our helm chart name itself is

wordpress.

So, if you go to into this folder, we are going to have all the files that we have discussed.

You can ignore the files with the name **helmignore**, **Chart.lock** and **values.schema.json**. So those are the files which are going to be maintained by the helm and they are not going to have any impact on us and we also do not have to create them manually.

So here let us explore the file which is **Chart.yaml**.

**C:\Users\niles\AppData\Local\Temp\helm\repository\wordpress-23.0.4\wordpress-23.0.4\wordpress\Chart.yaml**

So, if I try to open this with sublime text here, I have all the details like about my helm chart.

Like what is the API version?

apiVersion: v2

What is the app version?

appVersion: 6.6.0

What are the dependencies that my helm chart has?

So right now, my helm chart has dependency on other helm charts with the name Memcached, MariaDB and common.

dependencies:

- condition: memcached.enabled

  name: memcached

  repository: oci://registry-1.docker.io/bitnamicharts

  version: 7.x.x

- condition: mariadb.enabled

  name: mariadb

  repository: oci://registry-1.docker.io/bitnamicharts

  version: 19.x.x

- name: common

  repository: oci://registry-1.docker.io/bitnamicharts

And similarly, there is a description about this helm chart like ..

description: WordPress ...

And similarly, we have other metadata information like who is going to maintain them and what is the source of this helm chart and what is the version of this helm chart.

maintainers:

sources:

version: 23.0.4

So, all these details are available here.

So, these are all related to the metadata information about the helm chart.

So here let us explore the file which is **values.yaml**.

So let me also open the values.yaml file.

So, the values.yaml file will have all the dynamic values required for the templates that we have inside the helm chart.

So, these are all the key and value pairs.

And there are some values, there are some empty values.

So, all these values are going to be injected at runtime by the helm.

So here you can see they have given the Docker image details using which the deployment is going to

happen.

image:

  registry: docker.io

  repository: bitnami/wordpress

  tag: 6.6.0-debian-12-r0

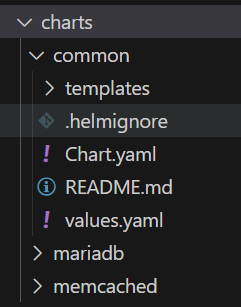
So here we have good number of values.

So, all these values are going to be used by the helm chart at runtime to prepare the Kubernetes manifest files based upon the templates that we have inside the templates folder.

So here let us explore the folder which is **charts**.

So now if you go and observe the charts folder inside this will be having all the dependent chart

details.



So right now, my chart is dependent on three other charts with the name common, mariadb and memcached.

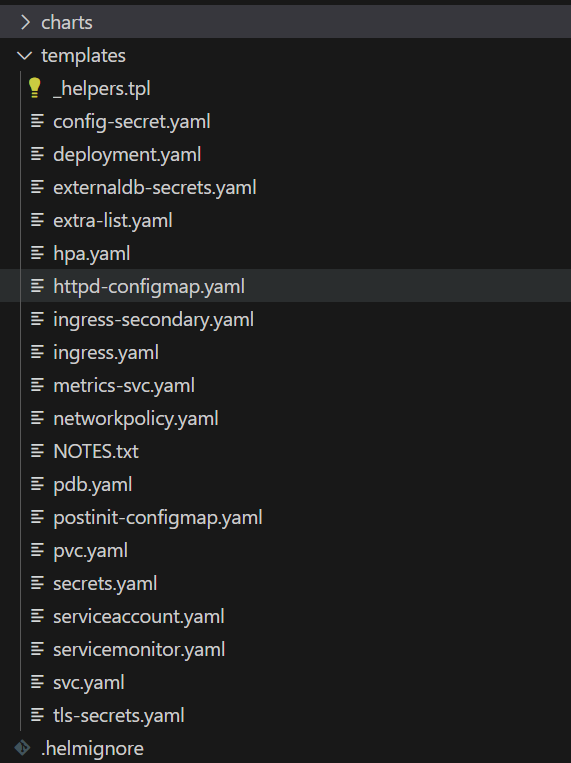
So, if you try to open any of these charts, they again will have the structure of helm chart because

at the end of the day, this mariadb, common and memcached also is a helm chart.

So here let us explore the folder which is **templates**.

So now if you go back to the parent folder here, we can go inside that templates folder, inside the

templates folder, you can see we have that **template files for all kind of required Kubernetes manifest files**.



Since we want to do the deployment of our microservices, there is a deployment related template file with the name deployment.yaml.

We can try to open the same.

So here I'm going to open the templates folder and open the deployment.yaml. Inside this deployment.yaml file you can see there is a template by following the standards of Kubernetes manifest file for the

kind: Deployment

Apart from the Kubernetes manifest file syntax, they are also trying to inject a lot many runtime dynamic values that they have defined inside these values.yaml file.

So please note that these values.yaml file will have values for all kind of template files apart from

deployment.yaml template file.

We also have many other template files.

So all these template files they are going to refer for the values defined inside these values.yaml.

With this, I am assuming your super clear about the structure of the helm charts, but as of now you

can see we installed a third-party helm chart which is already readily available. For our microservices

do you think a helm chart will be readily available?

Off course not.

We should build our own helm chart by following our own business requirements and Kubernetes manifest files.

Once we build a helm chart, we can deploy all our microservices using the same with a single command.

Always, we just need to maintain the **values.yaml** file for different, different environments for based

upon our requirements.

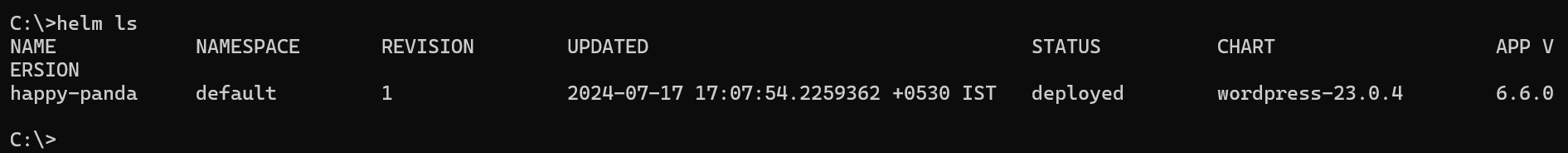
So as a next step, we need to create our own helm chart related to easy bank.

Before that, let me try to uninstall the wordpress related helm chart because it is already occupying

good amount of memory inside my Kubernetes cluster. I no longer need these wordpress helm chart.

So, before I try to uninstall, I am going to run the command which is

**> helm ls**

****

So, this is going to show you the list of installations that we have done with the help of Helm.

Like you can see here, we got an output.

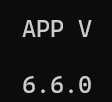
The name of the release is going to be Happy Panda.

And this is installed inside the default namespace and its revision is one and status is deployed and

the chart that we have used is wordpress and so on and so version and post that

we also have app version and this app version comes from the metadata information available inside that **Chart.yaml**

So, whatever this **app version** we can see here, this belongs to the wordpress app,



but whatever version you see here, this belongs to the helm chart, so here we are using the wordpress helm chart with the version 23.0.4.

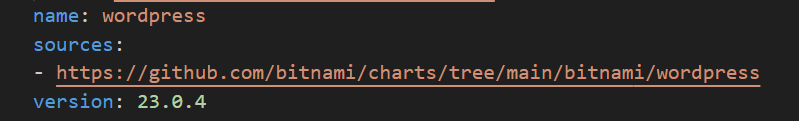


With this helm chart, we are trying to deploy a wordpress website which has a version of 6.6.0

**C:\Users\niles\AppData\Local\Temp\helm\repository\wordpress-23.0.4\wordpress-23.0.4\wordpress\Chart.yaml**

So, this app version information we can maintain inside the chart yaml you can go and check inside the Chart.yaml you will be able to see this version number.





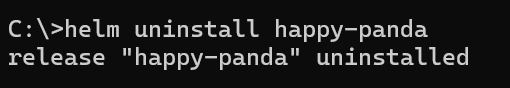
So now I have only one release installed with the help of helm and the same I want to uninstall.

So, what is the command to uninstall?

It is going to be super easy.

helm uninstall and what is your release name?

**> helm uninstall happy-panda**



So as soon as I execute this, the Uninstallation will complete.

Now if I go and check my Kubernetes dashboard here you can see there are no workloads to display inside my dashboard.

I can confirm by going to the deployment there are no deployments, there are no pods, there is no

replicaset and similarly, all services are deleted. And now I can also confirm inside the config maps those config maps related to wordpress are deleted and the same I can confirm under the secrets as well.

So, with a single helm uninstall command, I can uninstall my entire wordpress website.

So that is the power of helm chart.

Creating our own Helm chart and template files

As of now, we installed a helm chart which is available inside one of the public repositories with the

same helm chart, we set up the wordpress website, but in any real projects or in any enterprise organizations, they should build their own helm charts based upon their microservices requirements.

**Why we should build our own helm charts for our microservices?**

So that is why we should also try to build our own helm chart so that using the same helm chart, we

can deploy and uninstall all our microservices into the Kubernetes cluster with a single command.

And this will make our life very easy in maintaining any number of Kubernetes manifest files for any

number of microservices that we may have inside our organization.

So, to get started, I came to my workspace location, and here I am going to create **eazybank-helm** folder.

**D:\Experiments\Microservices\sb-bank-application\eazybank-helm**

I am going to create all the helm charts that I need for my microservices deployment.

So let me open my terminal at this location.

**PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm>**

So, to create the helm chart, we can run a command which is

***> helm create <name that we want to give for our helm chart>***

So, the name that I want to do here is **eazybank-common** because first I am going to build a **helm**

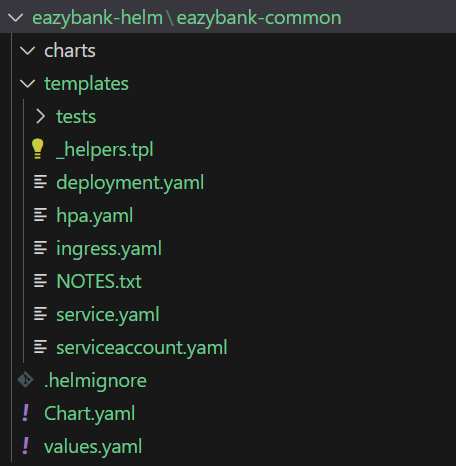
**chart that is going to be act as a common chart for all my microservices**.

So that is why I am giving this name.

**PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm> helm create eazybank-common**

So as soon as I enter inside my folder location, a helm chart might have created with the name

**eazybank-common**. We can validate the same here. So here I have a folder **eazybank-common** and inside this folder we have a helm chart with all the predefined files and folders that we have discussed previously.



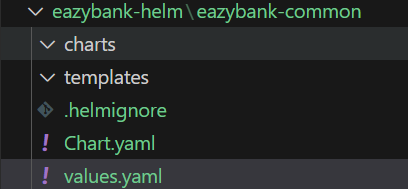
So, whenever we try to create a helm chart with the help of “helm create” command, the helm is going to give some predefined helm chart.

**DELETE ALL FILES INSIDE templates FOLDER**  
This predefined helm chart is going to have some content to deploy a website.

Since we want to write our own content, what we can do here is, first we can try to **delete all the template files that we have inside our helm chart**.

**DELETE ALL CONTENT INSIDE values.yaml**

And post that I will also open the values.yaml. So, inside the values.yaml we have many values related to nginx because using this nginx only the default helm chart is going to deploy a website. So, since we do not want to follow these values, **we can remove all the values**.



And as a next step I am going to check if there are any dependencies that this particular chart has and the **charts folder** you can see as of now it is empty, which means the default chart that is created by the Helm, it does not have any dependent helm charts.

So, **we should be good with this empty helm chart folder.**

As a next step, I will open this Chart.yaml. I am fine with the apiVersion, name, description and type as application and the version here is 0.1.0. So, this indicates what is the version of your helm chart. So, I am fine with the default value.

The next value here we have is app version. So, what is a version of app that you are trying to build? I want to maintain my own app version, so I will just keep it as 1.0.0.

**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-common\Chart.yaml**

appVersion: "1.0.0"

So, once we make these default changes as a next step, I am going to create the required template files for my eazybank microservices.

If you can recall in the previous section to deploy all our microservices, we have written **deployment,**

**manifest file** and **service manifest file** and **config map**.

As of now, my microservices need only these manifest files to deploy my microservices into the Kubernetes cluster.

So that is why I am going to create a three template files inside this helm chart which can be used by

all my microservices like accounts, loans, cards and many other microservices that I have inside

my microservice network.

**Copy the template files from**

[**https://github.com/eazybytes/microservices/tree/3.2.3/section\_16/helm/eazybank-common/templates**](https://github.com/eazybytes/microservices/tree/3.2.3/section_16/helm/eazybank-common/templates)

For the same, I am going to paste three template files inside this **templates folder** post that we can discuss all of them in detail. So here I am pasting few files.



Like you can see I have pasted three files, so let us try to open them and try to discuss them one by

one.

The first file that I want to discuss is about the service.yaml because this is simple yaml template file.

**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-common\templates\service.yaml**

**So first I am trying to define a name for this entire template, the name is common.service.**

{{- define "common.service" -}}

So, with the help of this, **define function or define keyword**, we are trying to give a name to this

entire Kubernetes template file so that other microservice helm charts that we are going to build in

future, they can try to refer with the name common.service.

So that is the purpose of define here.

And whatever **hyphens (-)** that you see at the starting and at the end, those are helpful to trim any space that you may have before and after of your statement.

*So basically, Helm is using Go template language and, in few places, it also uses spring template language. But we do not have to go deep dive into the helm because we are trying to learn helm on a high level, helm itself need a separate course to learn everything about helm. But we as a developer, we do not have to learn everything about Helm.*

So, whenever we are defining a name for your template with the help of this **define** at the end,

we should also close that by using this **end** statement.

{{- end -}}

So, whatever we have defined between the define and end, so this template will be assigned to a name which is common.service.

So now inside the template, the first element that I have is **apiVersion** with the value v1 and the

**kind** is Service. *So, these are static values. They are never going to change.* That is why I am directly mentioning the key and value for them.

apiVersion: v1

kind: Service

Now, under the **metadata** **name**, I need to inject a dynamic value.

So, whoever is going to use this template yaml file, they need to provide a serviceName inside their

values.yaml. The same name will get injected here.

So, **“Values”** are a helm framework object. So inside that values object keys and values that you have defined inside your values.yaml will be available and to access them.

I need to use “**.Values.**” and what is the key name of the property.

So, the key name is going to be “serviceName”

metadata:

  name: {{ .Values.serviceName }}

And similarly, under the **spec selector app** we are using a key which is “appLabel” inside the

Values object and the **type** for the service we are going to read with a key which is “service.type”

spec:

  selector:

    app: {{ .Values.appLabel }}

  type: {{ .Values.service.type }}

When I show you the values example for this template file, then it is going to make more sense to you. For now, please note that all the values that we have mentioned inside these curly braces along with the Values object, those are going to be dynamically injected at runtime by the helm based upon the values that we are going to provide.

Now under the **ports**, we have the **name** as “http” and the **protocol** is “TCP” and post that we need to provide what is the **port** and what is the **targetPort**.

Since these are dynamic values, I am trying to read them with the help of Values object with a key as

“service.port” and “service.targetPort”

  ports:

    - name: http

      protocol: TCP

      port: {{ .Values.service.port }}

      targetPort: {{ .Values.service.targetPort }}

So, I hope this is clear now very similarly, let me open the deployment template yaml file so that we

can discuss the same.

**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-common\templates\deployment.yaml**

Here also, we are using the same **define** and the **end**.

{{- define "common.deployment" -}}

apiVersion: apps/v1

kind: Deployment

metadata:

  name: {{ .Values.deploymentName }}

  labels:

    app: {{ .Values.appLabel }}

spec:

  replicas: {{ .Values.replicaCount }}

  selector:

    matchLabels:

      app: {{ .Values.appLabel }}

  template:

    metadata:

      labels:

        app: {{ .Values.appLabel }}

    spec:

      containers:

      - name: {{ .Values.appLabel }}

        image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"

        ports:

        - containerPort: {{ .Values.containerPort }}

          protocol: TCP

        env:

        {{- if .Values.appname\_enabled }}

        - name: SPRING\_APPLICATION\_NAME

          value: {{ .Values.appName }}

        {{- end }}

        {{- if .Values.profile\_enabled }}

        - name: SPRING\_PROFILES\_ACTIVE

          valueFrom:

            configMapKeyRef:

              name: {{ .Values.global.configMapName }}

              key: SPRING\_PROFILES\_ACTIVE

        {{- end }}

        {{- if .Values.config\_enabled }}

        - name: SPRING\_CONFIG\_IMPORT

          valueFrom:

            configMapKeyRef:

              name: {{ .Values.global.configMapName }}

              key: SPRING\_CONFIG\_IMPORT

        {{- end }}

        {{- if .Values.eureka\_enabled }}

        - name: EUREKA\_CLIENT\_SERVICEURL\_DEFAULTZONE

          valueFrom:

            configMapKeyRef:

              name: {{ .Values.global.configMapName }}

              key: EUREKA\_CLIENT\_SERVICEURL\_DEFAULTZONE

        {{- end }}

        {{- if .Values.resouceserver\_enabled }}

        - name: SPRING\_SECURITY\_OAUTH2\_RESOURCESERVER\_JWT\_JWK-SET-URI

          valueFrom:

            configMapKeyRef:

              name: {{ .Values.global.configMapName }}

              key: SPRING\_SECURITY\_OAUTH2\_RESOURCESERVER\_JWT\_JWK-SET-URI

        {{- end }}

        {{- if .Values.otel\_enabled }}

        - name: JAVA\_TOOL\_OPTIONS

          valueFrom:

            configMapKeyRef:

              name: {{ .Values.global.configMapName }}

              key: JAVA\_TOOL\_OPTIONS

        - name: OTEL\_EXPORTER\_OTLP\_ENDPOINT

          valueFrom:

            configMapKeyRef:

              name: {{ .Values.global.configMapName }}

              key: OTEL\_EXPORTER\_OTLP\_ENDPOINT

        - name: OTEL\_METRICS\_EXPORTER

          valueFrom:

            configMapKeyRef:

              name: {{ .Values.global.configMapName }}

              key: OTEL\_METRICS\_EXPORTER

        - name: OTEL\_SERVICE\_NAME

          value: {{ .Values.appName }}

        {{- end }}

        {{- if .Values.kafka\_enabled }}

        - name: SPRING\_CLOUD\_STREAM\_KAFKA\_BINDER\_BROKERS

          valueFrom:

            configMapKeyRef:

              name: {{ .Values.global.configMapName }}

              key: SPRING\_CLOUD\_STREAM\_KAFKA\_BINDER\_BROKERS

        {{- end }}

{{- end -}}

So, this is that deployment.yaml file.

As a next step, let me open the config map template that I have written.

So here we have a configmap.yaml.

{{- define "common.configmap" -}}

apiVersion: v1

kind: ConfigMap

metadata:

  name: {{ .Values.global.configMapName }}

data:

  SPRING\_PROFILES\_ACTIVE: {{ .Values.global.activeProfile }}

  SPRING\_CONFIG\_IMPORT: {{ .Values.global.configServerURL }}

  EUREKA\_CLIENT\_SERVICEURL\_DEFAULTZONE: {{ .Values.global.eurekaServerURL }}

  SPRING\_SECURITY\_OAUTH2\_RESOURCESERVER\_JWT\_JWK-SET-URI: {{ .Values.global.keyCloakURL }}

  JAVA\_TOOL\_OPTIONS: {{ .Values.global.openTelemetryJavaAgent }}

  OTEL\_EXPORTER\_OTLP\_ENDPOINT: {{ .Values.global.otelExporterEndPoint }}

  OTEL\_METRICS\_EXPORTER: {{ .Values.global.otelMetricsExporter }}

  SPRING\_CLOUD\_STREAM\_KAFKA\_BINDER\_BROKERS: {{ .Values.global.kafkaBrokerURL }}

{{- end -}}

But I'm not trying to mention the direct hardcoded values inside the configmap.yaml because I may

have different, different requirement like for dev environment, I may have different profile, different URLs.

And similarly for QA and prod, I may have different values.

So that's why we are going to maintain values.yaml for different, different environments.

But the template is going to be the same.

That is why I am trying to mention these variable names instead of hardcoded values inside this template file.

So, with this I am assuming you are clear about these three template Yaml files that I have written inside the eazybank-common helm chart.

**D:\Experiments\Microservices\sb-bank-application\eazybank-helm-practice\eazybank-common\values.yaml**

**Why we will keep the values.yaml for this chart empty?**  
As a next step we can check values.yaml and make sure this is empty because this helm chart, which

is eazybank-common, this is going to be leveraged by the other helm charts.

So whoever is going to leverage this common helm chart, they are going to provide their own values.yaml and this helm chart, which is eazybank-common does not have any specific requirements to pass its own values.

**So that's why we are going to keep the values.yaml for this helm chart as empty.**

So, with this eazybank -common helm chart, we have prepared a common helm chart which has a template files related to service, deployment, and config map.

As a next step, we need to create the helm charts which can leverage these templates and provide them own values.yaml based upon the microservices requirement.

Creating Helm chart for Accounts microservice

Inside this lecture, I am going to show you how to create a helm chart for our accounts

microservice by leveraging the **eazybank-common** helm chart.

Because inside this eazybank-common only, we have defined all the required Kubernetes manifest template files.

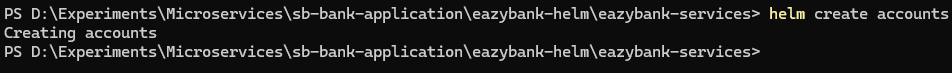
So, my accounts microservice it must use these templates and it should also mention the values inside the values.yaml.

For the same, I am going to create a new folder here with the name **eazybank-services** because inside this folder I am going to create all the helm charts that are required for my microservices.

**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services**

Inside this folder only, I am going to create a new helm chart with the name accounts.

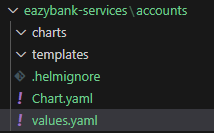
**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services> helm create accounts**



Inside my **eazybank-services**, I have a new folder with the name **accounts**. Like we discussed before,

first, we need to delete all the templates that we received from the default helm chart.

And very similarly I will also delete the values.yaml content.



**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\accounts\Chart.yaml**

apiVersion: v2

name: accounts

description: A Helm chart for Kubernetes

type: application

version: 0.1.0

appVersion: "1.0.0"

The only thing we will change is the **appVersion** to 1.0.0, rest all is fine.

So now inside the same Chart.yaml, I need to define if this helm chart has any dependency on other helm charts.

We know that this helm chart which is accounts microservice related helm chart has a dependency on the eazybank-common helm chart.

So how to define those details, for the same just after the app version we mention a new element which is **dependencies**.

And inside this dependencies we mention the list of helm chart that this particular chart has depends on.

**name:**   
So here first we need to make sure we are mentioning the **name** of the helm chart, which is eazybank-common.

**version:**  
After that please mention the **version** of helm chart of eazybank-common.

If you go and check the eazybank-command and open this Chart.yaml it has the version which

is 0.1.0 the same version we need to mention here also.

**repository:**

And after the version we need to mention **repository** element, and to this repository we need to provide the path where this eazybank-common helm chart is available.   
- If your helm chart is available inside a URL, you can mention directly the URL details.   
- Since my helm chart is available inside my local system,   
 *I need to mention this file post that colon followed by two forward slash and I need to mention  
 these two dots and I am going to repeat the two dots one more time*

**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\accounts\Chart.yaml**

...

dependencies:

  - name: eazybank-common

    version: 0.1.0

    repository: <file://../../eazybank-common>

...

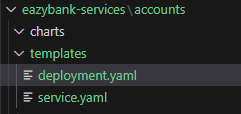
So, once we have defined these dependencies, we should be good.

As a next step, I need to provide those templates inside the accounts helm chart.

So as of now my templates folder is empty.

**Here you may have a question like we already defined those templates inside the eazybank-common, then why should we define again here?**

I am going to answer that question in few seconds, but before that let me copy the template files into this folder location.



**Copy the template files from**

[**https://github.com/eazybytes/microservices/blob/3.2.3/section\_16/helm/eazybank-services/accounts/templates/deployment.yaml**](https://github.com/eazybytes/microservices/blob/3.2.3/section_16/helm/eazybank-services/accounts/templates/deployment.yaml)

[**https://github.com/eazybytes/microservices/blob/3.2.3/section\_16/helm/eazybank-services/accounts/templates/service.yaml**](https://github.com/eazybytes/microservices/blob/3.2.3/section_16/helm/eazybank-services/accounts/templates/service.yaml)

**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\accounts\templates\deployment.yaml**

{{- template "common.deployment" . -}}

**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\accounts\templates\service.yaml**

{{- template "common.service" . -}}

So here I have pasted two template files with the name deployment.yaml and service.yaml because

to deploy my accounts microservice, I just need to apply the deployment manifest file and service manifest file.

I do not have any other requirements, if you have other requirements where you need to define other Kubernetes objects, then definitely you need to define the template files specific to those Kubernetes objects.

So, if you try to open this Yaml files and try to understand what is there, you will get your answer.

So here I'm just trying to refer other template that I have defined with the name common.deployment

{{- template "common.deployment" . -}}

Do you recall where we have defined this template inside the eazybank-common helm chart?

We have defined this template with the name common.deployment, so I'm simply trying to refer to

the template available inside the other helm chart, which is eazybank-common.

**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-common\templates\deployment.yaml**

So that is why first we need to make sure we added the dependencies and post that only we should be able to use this template.

And what is the name of the template by following the syntax.

**Please make sure this dot is also present because this is a syntax that we need to follow.**

So now very similarly, if I go and observe the service.yaml template file, the same kind of setup

will be there.

{{- template "common.service" . -}}

Like where I'm trying to refer the other template with the name common.service.

**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-common\templates\service.yaml**

So, with this I am assuming your question is answered.

So as a next step, we need to populate all the required values inside the values.yaml of accounts

microservice.

So basically define all values defined in deployment.yaml and service.yaml in eazybank-common helm chart with **“.Values.”**

**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\accounts\values.yaml**

**Important Note:  
Format 1:  
name: {{ .Values.deploymentName }} 🡺 template definition**

**below is the values definition**

deploymentName: accounts-deployment

**Format 2:**

**image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}" 🡺 template definition  
below is the values definition**

image:

  repository: eazybytes/accounts

  tag: s14

deploymentName: accounts-deployment

serviceName: accounts

appLabel: accounts

appName: accounts

replicaCount: 1

image:

  repository: eazybytes/accounts

  tag: s14

containerPort: 8080

service:

  type: ClusterIP

  port: 8080

  targetPort: 8080

appname\_enabled: true

profile\_enabled: true

config\_enabled: true

eureka\_enabled: true

resouceserver\_enabled: false

otel\_enabled: true

kafka\_enabled: true

So whatever value we have mentioned here, this will be mapped to the service.yaml and deployment.yaml file that we have defined inside the templates folder of eazybank-common chart.

**Compiling the helm charts**

Here you can see the chart folder is empty.

Previously I said all the dependent helm charts will be available inside the chart folder, but right

now it is empty because we have not compiled our helm chart as of now.

So, for that reason we are not able to see the dependent helm chart details inside the accounts microservice.

So as a next step, let us try to compile this helm chart for the same inside our terminal, we need to go into the helm chart where we need to compile.

So, the helm chart name is accounts.

Right now, I am into the helm chart location.

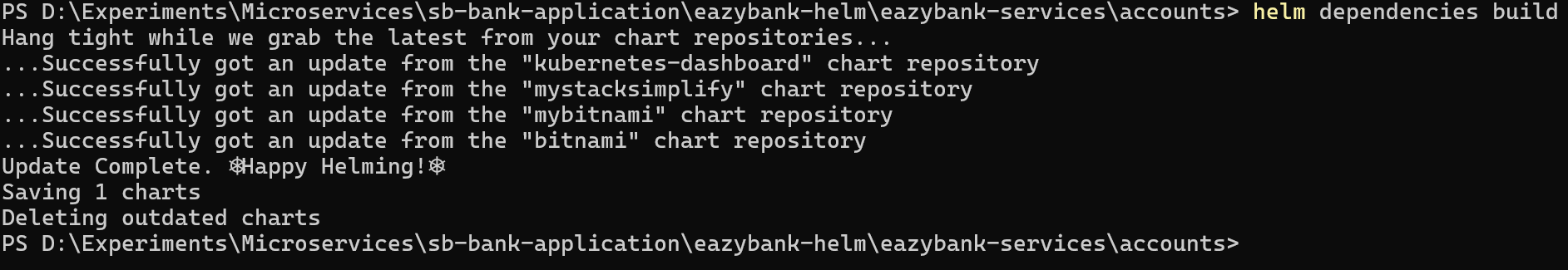
**PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\accounts>**

Here I am going to run the command, which is

**PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\accounts> helm dependencies build**

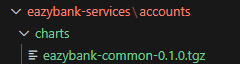
So, this helm dependencies build will compile my helm chart accounts and it will also try to compile all the dependent helm charts and place them inside the chart folder.

So let me try to execute this command.



You can see we are getting an output saying that the compilation is successful.

Let us go to the folder and try to validate the same.



So right now, if I go and observe the chart folder, there will be a compressed helm chart with the

name **eazybank-common-0.1.0.tgz**.

So, this is how the dependent helm chart is going to come into your own helm chart.

So always make sure you are running the helm dependencies build command whenever your helm chart has dependency on another helm chart.

For eazybank-common we did not run this command because the easy bank command does not have dependency on any other helm charts.

So, with this I am assuming your super clear on how to create a helm chart for a specific microservice.

Creating Helm chart for other microservices

**Steps to create cards helm charts**

**Step 1: Creating helm charts**

Go inside folder

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services

Inside this folder only, I am going to create a new helm chart with the name cards.

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services> helm create cards

**Step 2: Clean up**Delete all files inside templates folder and empty the values.yaml file

**Step 3: Update Charts.yaml application version**

The only thing we will change is the appVersion to 1.0.0, rest all is fine.

**Step 4: Add Chart dependencies in Chart.yaml**

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\cards\Chart.yaml

**Step 5: Copy the template files**

Copy the template files from

<https://github.com/eazybytes/microservices/blob/3.2.3/section_16/helm/eazybank-services/accounts/templates/deployment.yaml>

<https://github.com/eazybytes/microservices/blob/3.2.3/section_16/helm/eazybank-services/accounts/templates/service.yaml>

**Step 6: Update the values.yaml**

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\cards\values.yaml

Double check the key-value pair values are specific to the service, refer the docker-compose files if required. Make sure you have covered each property from the deployment.yaml and service.yaml from the template in the cards or common chart.

**Step 7: Compile the helm chart**

Compiling the helm charts

PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\cards>

PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\cards> helm dependencies build

**Steps to create loans helm charts**

Go inside folder

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services

Inside this folder only, I am going to create a new helm chart with the name loans.

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services> helm create loans

**Step 2: Clean up**Delete all files inside templates folder and empty the values.yaml file

**Step 3: Update Charts.yaml application version**

The only thing we will change is the appVersion to 1.0.0, rest all is fine.

**Step 4: Add Chart dependencies in Chart.yaml**

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\loans\Chart.yaml

**Step 5: Copy the template files**

Copy the template files from

<https://github.com/eazybytes/microservices/blob/3.2.3/section_16/helm/eazybank-services/accounts/templates/deployment.yaml>

<https://github.com/eazybytes/microservices/blob/3.2.3/section_16/helm/eazybank-services/accounts/templates/service.yaml>

**Step 6: Update the values.yaml**

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\loans\values.yaml

Double check the key-value pair values are specific to the service, refer the docker-compose files if required. Make sure you have covered each property from the deployment.yaml and service.yaml from the template in the loans or common chart.

**Step 7: Compile the helm chart**

Compiling the helm charts

PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\loans>

PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\loans> helm dependencies build

**Steps to create configserver helm charts**

Go inside folder

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services

Inside this folder only, I am going to create a new helm chart with the name configserver.

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services> helm create configserver

**Step 2: Clean up**Delete all files inside templates folder and empty the values.yaml file

**Step 3: Update Charts.yaml application version**

The only thing we will change is the appVersion to 1.0.0, rest all is fine.

**Step 4: Add Chart dependencies in Chart.yaml**

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\configserver\Chart.yaml

**Step 5: Copy the template files**

Copy the template files from

<https://github.com/eazybytes/microservices/blob/3.2.3/section_16/helm/eazybank-services/accounts/templates/deployment.yaml>

<https://github.com/eazybytes/microservices/blob/3.2.3/section_16/helm/eazybank-services/accounts/templates/service.yaml>

**Step 6: Update the values.yaml**

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\configserver\values.yaml

Double check the key-value pair values are specific to the service, refer the docker-compose files if required. Make sure you have covered each property from the deployment.yaml and service.yaml from the template in the configserver or common chart.

**Step 7: Compile the helm chart**

Compiling the helm charts

PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\configserver>

PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\configserver> helm dependencies build

**Steps to create eureka helm charts**

Go inside folder

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services

Inside this folder only, I am going to create a new helm chart with the name eureka.

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services> helm create eureka

**Step 2: Clean up**Delete all files inside templates folder and empty the values.yaml file

**Step 3: Update Charts.yaml application version**

The only thing we will change is the appVersion to 1.0.0, rest all is fine.

**Step 4: Add Chart dependencies in Chart.yaml**

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\eureka\Chart.yaml

**Step 5: Copy the template files**

Copy the template files from

<https://github.com/eazybytes/microservices/blob/3.2.3/section_16/helm/eazybank-services/accounts/templates/deployment.yaml>

<https://github.com/eazybytes/microservices/blob/3.2.3/section_16/helm/eazybank-services/accounts/templates/service.yaml>

**Step 6: Update the values.yaml**

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\eureka\values.yaml

Double check the key-value pair values are specific to the service, refer the docker-compose files if required. Make sure you have covered each property from the deployment.yaml and service.yaml from the template in the eureka or common chart.

**Step 7: Compile the helm chart**

Compiling the helm charts

PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\eureka>

PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\eureka> helm dependencies build

**Steps to create gatewayserver helm charts**

Go inside folder

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services

Inside this folder only, I am going to create a new helm chart with the name gatewayserver.

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services> helm create gatewayserver

**Step 2: Clean up**Delete all files inside templates folder and empty the values.yaml file

**Step 3: Update Charts.yaml application version**

The only thing we will change is the appVersion to 1.0.0, rest all is fine.

**Step 4: Add Chart dependencies in Chart.yaml**

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\gatewayserver\Chart.yaml

**Step 5: Copy the template files**

Copy the template files from

<https://github.com/eazybytes/microservices/blob/3.2.3/section_16/helm/eazybank-services/accounts/templates/deployment.yaml>

<https://github.com/eazybytes/microservices/blob/3.2.3/section_16/helm/eazybank-services/accounts/templates/service.yaml>

**Step 6: Update the values.yaml**

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\gatewayserver\values.yaml

Double check the key-value pair values are specific to the service, refer the docker-compose files if required. Make sure you have covered each property from the deployment.yaml and service.yaml from the template in the gatewayserver or common chart.

**Step 7: Compile the helm chart**

Compiling the helm charts

PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\gatewayserver>

PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\gatewayserver> helm dependencies build

**Steps to create message helm charts**

Go inside folder

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services

Inside this folder only, I am going to create a new helm chart with the name message.

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services> helm create message

**Step 2: Clean up**Delete all files inside templates folder and empty the values.yaml file

**Step 3: Update Charts.yaml application version**

The only thing we will change is the appVersion to 1.0.0, rest all is fine.

**Step 4: Add Chart dependencies in Chart.yaml**

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\message\Chart.yaml

**Step 5: Copy the template files**

Copy the template files from

<https://github.com/eazybytes/microservices/blob/3.2.3/section_16/helm/eazybank-services/accounts/templates/deployment.yaml>

<https://github.com/eazybytes/microservices/blob/3.2.3/section_16/helm/eazybank-services/accounts/templates/service.yaml>

**Step 6: Update the values.yaml**

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\message\values.yaml

Double check the key-value pair values are specific to the service, refer the docker-compose files if required. Make sure you have covered each property from the deployment.yaml and service.yaml from the template in the message or common chart.

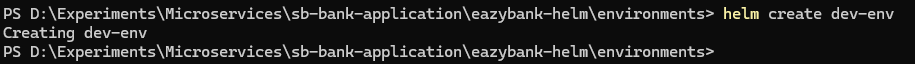
**Step 7: Compile the helm chart**

Compiling the helm charts

PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\message>

PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm\eazybank-services\message> helm dependencies build

Creating Helm Chart for Dev, Qa and Prod environment

Create a helm chart for dev-env 

Delete all files inside folder

**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\environments\dev-env\templates**

Delete all content inside file

**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\environments\dev-env\values.yaml**

Modify the appVersion of file

**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\environments\dev-env\Chart.yaml**

appVersion: "1.0.0"

Add dependencies of all microservices in file

**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\environments\dev-env\Chart.yaml**

...

appVersion: "1.0.0"

dependencies:

  - name: eazybank-common

    version: 0.1.0

    repository: file://../../eazybank-common

  - name: configserver

    version: 0.1.0

    repository: file://../../eazybank-services/configserver

  - name: eurekaserver

    version: 0.1.0

    repository: file://../../eazybank-services/eurekaserver

  - name: accounts

    version: 0.1.0

    repository: file://../../eazybank-services/accounts

  - name: cards

    version: 0.1.0

    repository: file://../../eazybank-services/cards

  - name: loans

    version: 0.1.0

    repository: file://../../eazybank-services/loans

  - name: gatewayserver

    version: 0.1.0

    repository: file://../../eazybank-services/gatewayserver

  - name: message

    version: 0.1.0

    repository: file://../../eazybank-services/message

Copy the configmap.yaml from   
<https://github.com/eazybytes/microservices/blob/3.2.3/section_16/helm/environments/dev-env/templates/configmap.yaml>

To folder   
**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\environments\dev-env\templates**

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\environments\dev-env\templates\configmap.yaml

{{- template "common.configmap" . -}}

Add properties to the values.yaml the

**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\environments\dev-env\values.yaml**

global:

  configMapName: eazybankdev-configmap

  activeProfile: default

  configServerURL: configserver:http://configserver:8071/

  eurekaServerURL: http://eurekaserver:8070/eureka/

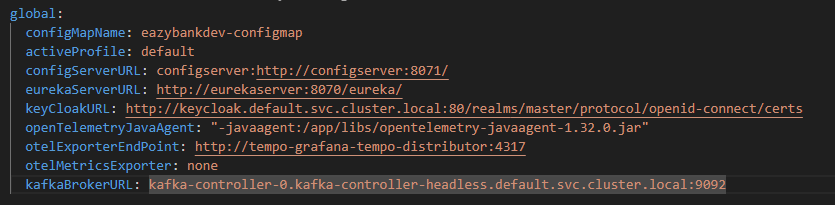
  keyCloakURL: http://keycloak.default.svc.cluster.local:80/realms/master/protocol/openid-connect/certs

  openTelemetryJavaAgent: "-javaagent:/app/libs/opentelemetry-javaagent-1.32.0.jar"

  otelExporterEndPoint: http://tempo-grafana-tempo-distributor:4317

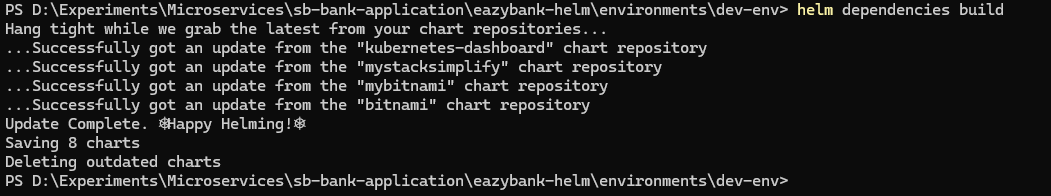
  otelMetricsExporter: none

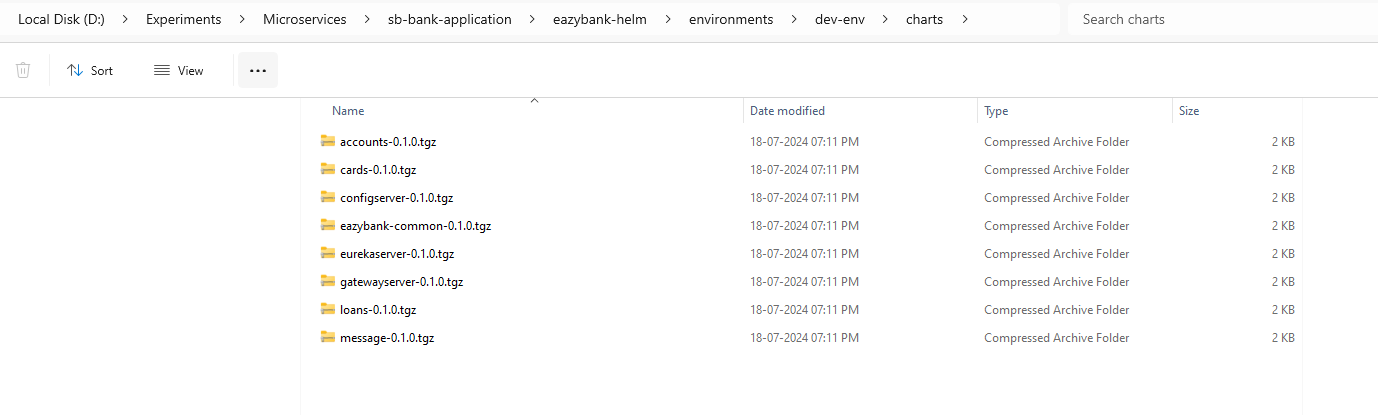
  kafkaBrokerURL: kafka-controller-0.kafka-controller-headless.default.svc.cluster.local:9092

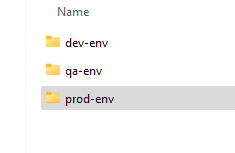


Compile the helm charts

**PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm\environments\dev-env> helm dependencies build**







Update profile

**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\environments\qa-env\values.yaml**

 activeProfile: qa

Update profile

**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\environments\prod-env\values.yaml**

 activeProfile: prod

Demo of “helm template” command

As of now, we have done a lot of pre-work on preparing our own helm charts, which are specific to

eazybank microservice.

Sometimes before you try to install your helm charts, you may want to know how your Kubernetes manifest files are going to look like that are going to be generated by your helm chart.

So, to do the same validation, we have a command inside the helm.

So, the command is   
**> helm template**

So, with the help of this “helm template” command, we can see all the Kubernetes files that are going to be generated by your helm.

And the same Kubernetes manifest files are also going to be installed inside your Kubernetes cluster.

So, it will be a wise decision to validate the Kubernetes manifest files before we try to install them

into the Kubernetes cluster.

So, to execute this helm template command, we need to make sure our terminal is opened inside your helm chart.

As of now, you can see I am into the folder which is dev-env.

PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm\environments\dev-env> **helm template .**

So, from this location I can try to run the command which is helm template space dot.

So, with this dot I am telling my helm template is available in this folder only.

Please access the same and show me all the Kubernetes manifest files that you are going to prepare.

So, I am trying to execute this.

You can expect a big output of all our Kubernetes manifest files that it is going to generate for our microservices.

I am sure it is going to work.

So as a next step to deploy all our microservices into the Kubernetes cluster, we can simply install

one of the environment helm charts.

But before that we need to make sure we also set up, keycloak, kafka and grafana related components inside our Kubernetes cluster.

For those, we do not have to prepare any helm charts manually because all these products are heavily used by the industry.

And with that reason, many open-source communities, and organizations they built the helm charts specific to these components like Kafka, Keycloak and Grafana.

So, let us try to explore them and try to learn how to set up those components, with the help of Helm.

Install Keycloak in Kubernetes Cluster using Helm Chart

One of the great advantages of Helm is it has a very good community.

With that reason, you should be able to easily find the helm charts to install any product inside the

software industry.

For example, if you want to install Kafka inside your Kubernetes cluster, you do not have to prepare

the Kubernetes manifest files manually.

Instead, you can rely on the helm charts available inside the web.

*So, one such company or community, which always maintain these helm charts with a great production standard is Bitnami.*

*So bitnami make it easy to get your favourite open-source software’s up and running on any platform, including your laptop, Kubernetes or all the major clouds.*

*So, this bitnami is supported by VMware.*

*So, inside this course we are going to leverage the Bitnami Helm charts to install all the products like*

*Kafka, Grafana, Prometheus, Keycloak inside our Kubernetes cluster.*

For the same inside the Google, you can search for bitnami helm charts GitHub.

This will show us the GitHub repo where Bitnami is maintaining all the helm charts.

<https://github.com/bitnami/charts/tree/main/bitnami>

*Inside this repo, if you can open this Bitnami folder, you should be able to identify the helm charts*

*for most of the products that any organization needs.*

*So here we have products like Argo CD, Cassandra Console, followed by Drupal, Elasticsearch.*

*Similarly, all Grafana related helm charts are also available here like Grafana Loki Grafana Tempo,*

*Grafana. If you can scroll down, you will be able to see many other products like Kafka, Keycloak, Kibana,*

*Cube prometheus, MongoDB, logstash, Mariadb. Similarly, you should be able to identify some helm charts related to MySQL, Nginx, Postgres database, Rabbitmq, Redis.*

So, there are good amount of helm charts available.

So based upon your requirements you can always leverage these Bitnami helm charts.

As of now we leverage one of the helm charts which is wordpress while seeing demo of helm charts

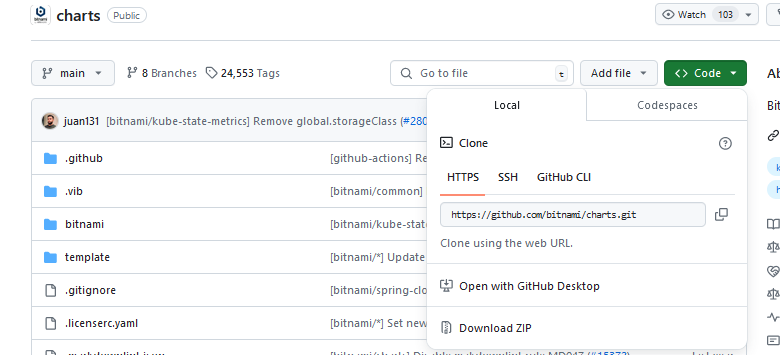
in the previous lectures.

So, in the same location we have many other helm charts and we are going to leverage them to set up required components inside our Kubernetes cluster.

So, to download these helm charts, what I can do is, I can go to the charts folder and here we have an option to download all these helm charts into local system.

So, I am going to click on this download Zip option and this will download all the helm charts that are

provided by the Bitnami.



I can choose whatever I required and leverage them to install the components inside my Kubernetes cluster.

Let me go to the downloads folder and look for the helm charts that we have downloaded. Inside my downloads folder, like you can see I have a folder with the name Bitnami.

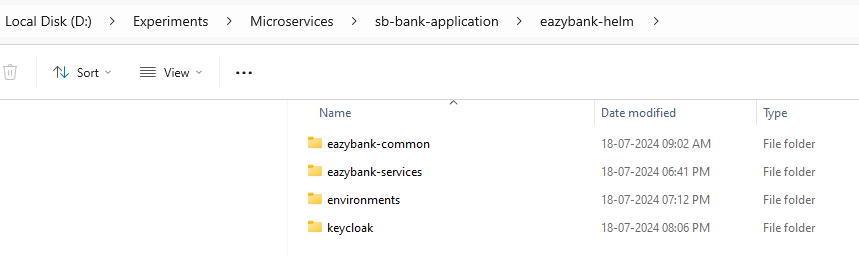
C:\Users\niles\Downloads\charts-main\charts-main\bitnami

If I can open inside this, I have all the helm charts that are provided by the Bitnami.

As a first step, I'm going to use the Keycloak Helm chart from the bitnami folder

C:\Users\niles\Downloads\charts-main\charts-main\bitnami\keycloak

And paste it in   
D:\Experiments\Microservices\sb-bank-application\eazybank-helm



Inside this Keycloak Helm chart we have all the required files like Chart.yaml, templates and values.yaml.

If I try to run this Keycloak helm chart into my local Kubernetes cluster by default it is going to

deploy my keycloak service with a cluster IP.

But since I want to access my keyclock to create the client details and roles information, I am going

to expose my key clock service as a load balancer.

So to change this behaviour, we need to open the values.yaml present inside the keycloak helm chart.

Search for

type: ClusterIP

So here instead of cluster IP, I am going to mention as LoadBalancer, once you have defined the service type as LoadBalancer,

type: LoadBalancer

you can look for

 ##

  adminUser: user

  ## @param auth.adminPassword Keycloak administrator password for the new user

  ##

  adminPassword: ""

So here I'm going to give the password as password itself so that we can login into the keycloak admin with the username as user and the password as password.

adminPassword: "password"

So, once we make these changes, we should be good.

Let me go to that terminal. Inside my terminal, I am going to install the helm chart of keycloak.

PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm\keycloak**> helm dependencies build**

For the same, let me go to the location where my key clock helm chart is available.

**PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm>**

D:\Experiments\Microservices\sb-bank-application\eazybank-helm\keycloak\Chart.yaml

name: keycloak

So let me try to execute this command.

*PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm> helm install keycloak keycloak*

*Error: INSTALLATION FAILED: cannot re-use a name that is still in use*

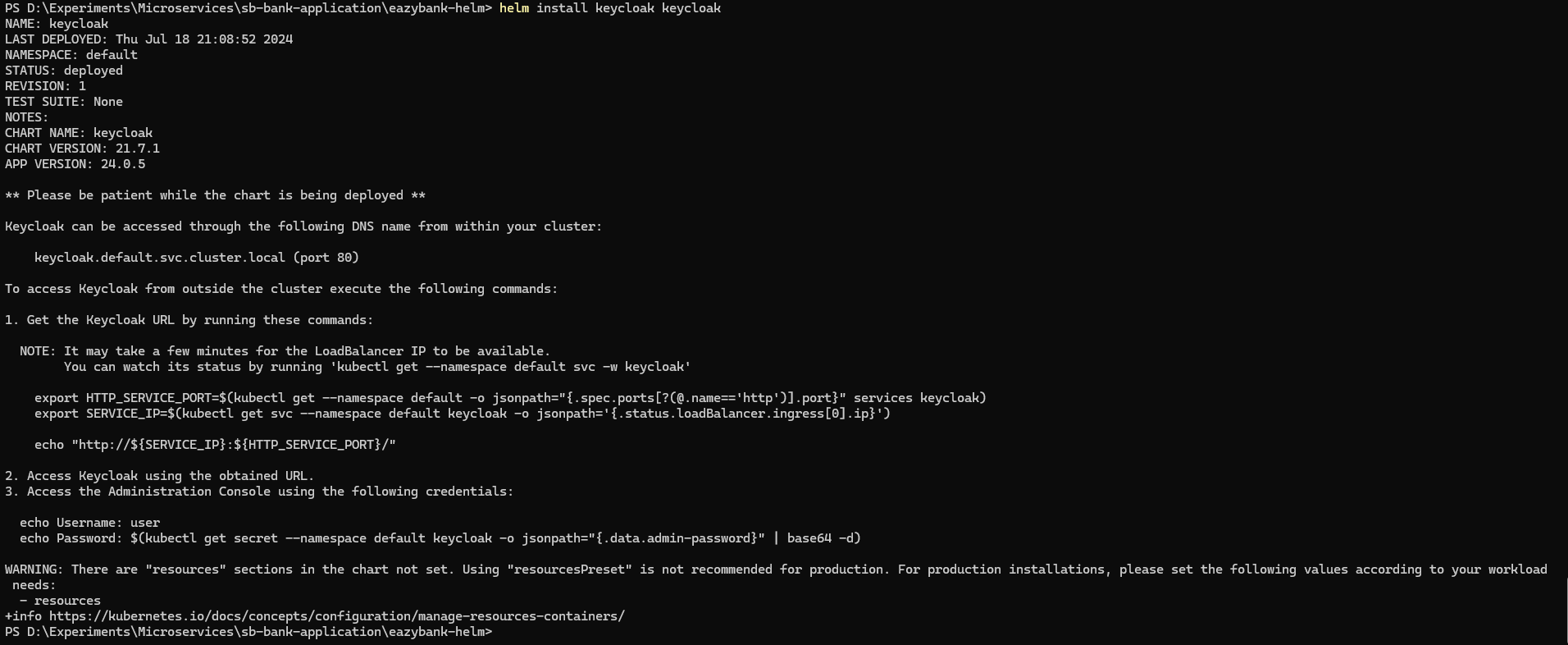
*PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm> helm uninstall keycloak*

*release "keycloak" uninstalled*

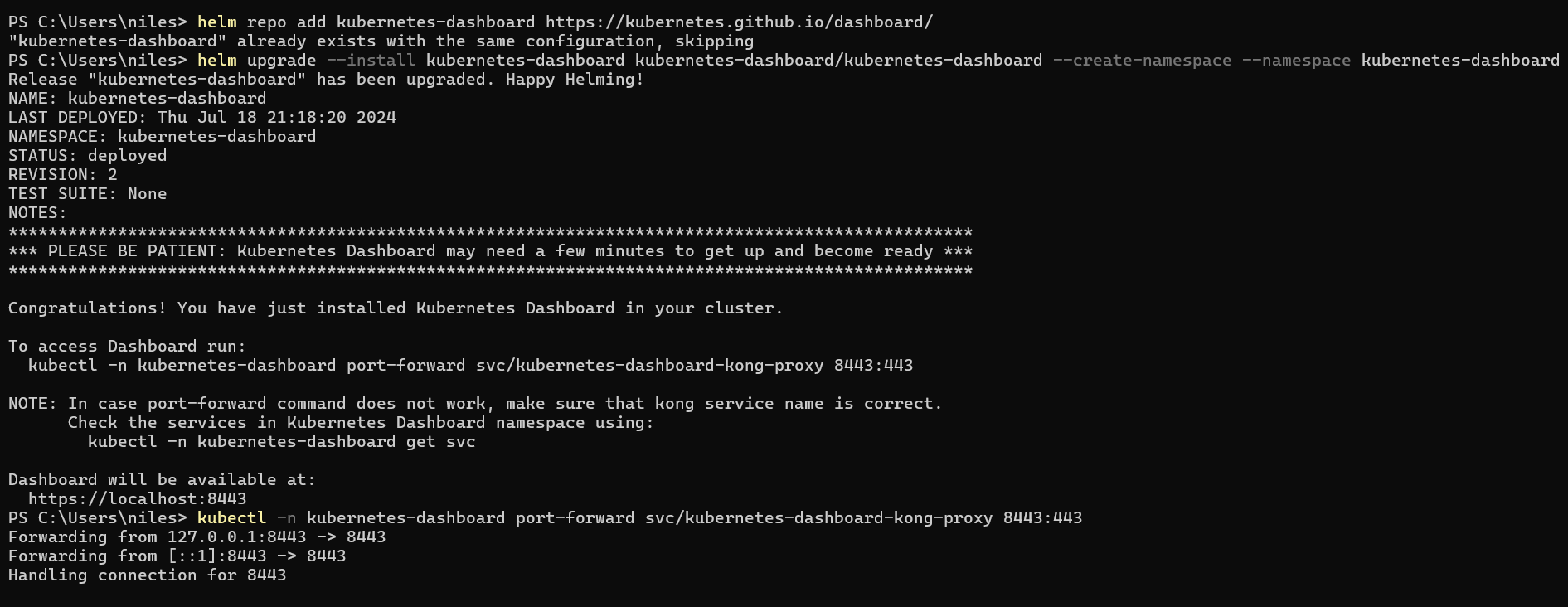
PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm**> helm install keycloak keycloak**

JUST MAKE SURE YOU USE NEW RELEASSE NAMES EACH TIME AND DELETE OLD Kubernetes resources from dashboard

And behind the scenes you can see the installation is completed.



Steps to start the Kubernetes dashboard



And I also got an output saying that what are the steps that I can follow to access my keycloak URL?

niles@Nilesh MINGW64 ~

$ export SERVICE\_IP=$(kubectl get svc --namespace default keycloak -o jsonpath='{.status.loadBalancer.ingress[0].ip}')

niles@Nilesh MINGW64 ~

$ export HTTP\_SERVICE\_PORT=$(kubectl get --namespace default -o jsonpath="{.spec.ports[?(@.name=='http')].port}" services keycloak)

niles@Nilesh MINGW64 ~

$ echo "http://${SERVICE\_IP}:${HTTP\_SERVICE\_PORT}/"

http://:80/

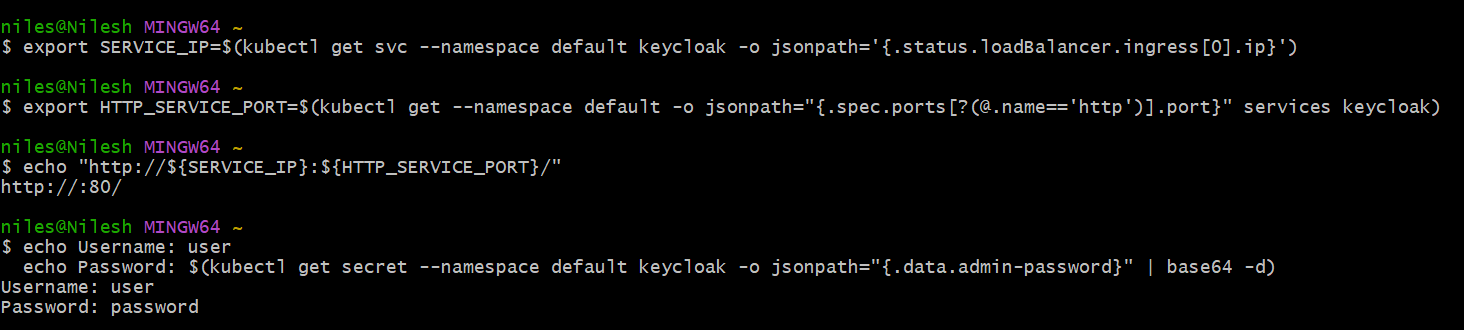
niles@Nilesh MINGW64 ~

$ echo Username: user

echo Password: $(kubectl get secret --namespace default keycloak -o jsonpath="{.data.admin-password}" | base64 -d)

Username: user

Password: password



Here you can see I got an output saying that I can able to access my keycloak at the URL http and hostname.

Since it is empty we can use localhost and the port is 80.

And similarly, if we want to know what is the username and password, there is also instructions on

how to know the username and password.

You can try to take these commands.

Let me try to clean the console and run the commands again.

So the username is user and the password is going to be the password because inside the values.yaml we have provided the password as password itself.

Please wait for 1 to 2 minutes for the LoadBalancer to get successfully created inside your local system. It is going to take easily 1 to 2 minutes at least.

Post that you should be able to access the keycloak at the URL <http://localhost>

So let me try to access the same. So here I am trying to access the URL. You can see I am able to access my administration console.

So let me click on that post that the username I am going to enter as user and the password is going

to be password.

After entering the credentials, I am clicking on the sign in button.

Create client and roles inside keycloak.

Did you see how easy it is to set up a Keycloak server with the production standards by using Helm Chat? That is a power of helm chats.

If you do not have helm charts, then definitely you need to prepare a lot many Kubernetes manifest files.

I can also show you the template files, and with that you should be able to understand how complex

to create and install Keycloak inside a production Kubernetes cluster.

So here under the templates, you should be able to see lot many templates related to Kubernetes objects.

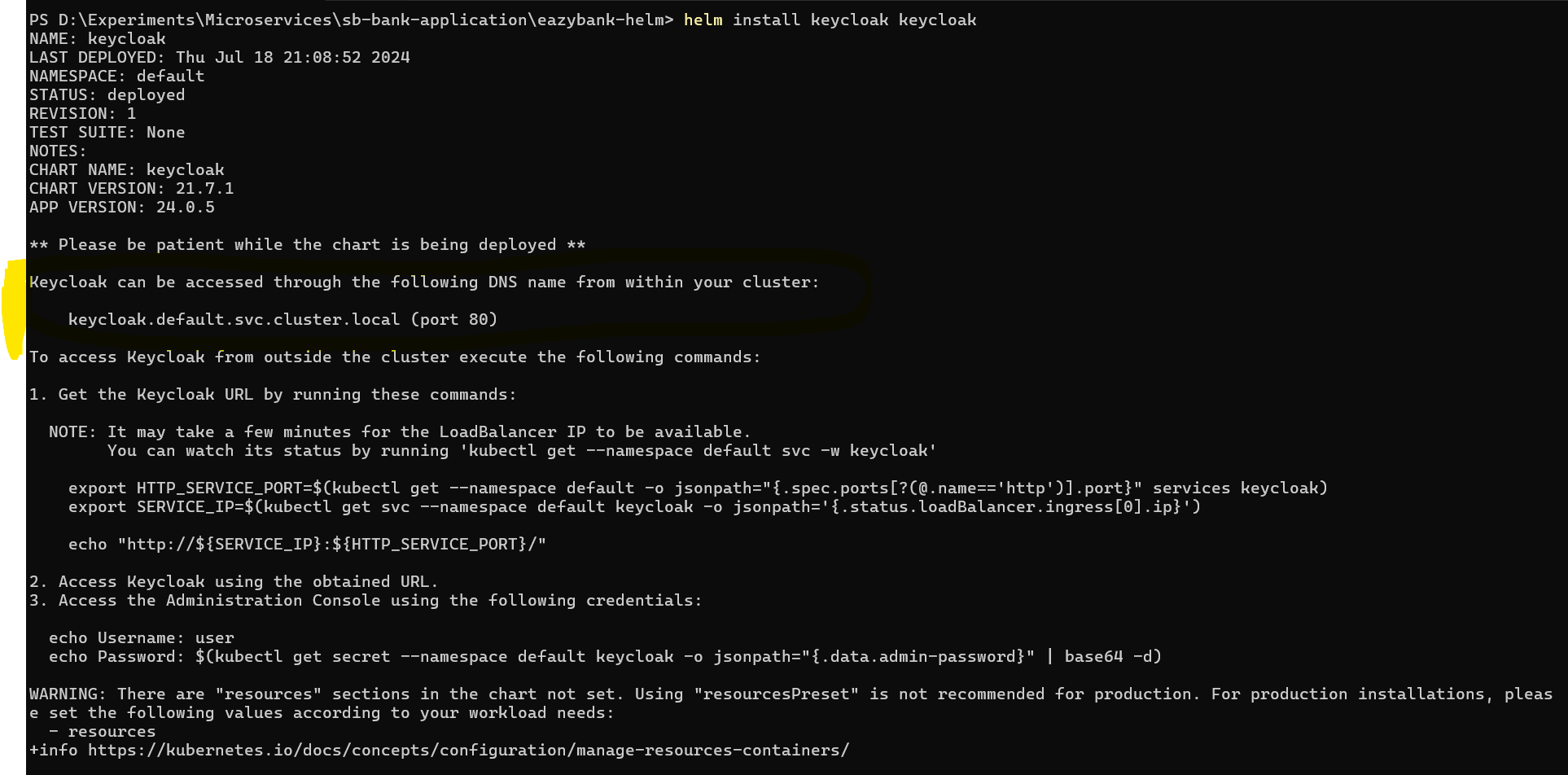
So, all these to create manually is going to be a super cumbersome process.

On top of that, my keycloak also has dependency on other helm charts like Postgres because behind the scenes my keycloak is going to use Postgres database.

As a next step, I want you to explain how we can derive the keycloak endpoint URL that we can feed to the other microservices.

If you see here inside the values.yaml of all the environments we have mentioned, the keycloak URL

as this is the host name, this is the port number.



So how I know what is a hostname that I want to provide.

For the same, if you try to look at the output console that we received from the helm installation command here, there is a important information saying that keycloak can be accessed to the following DNS name from within your cluster.

So this is the DNS name that they have provided the same name I have mentioned inside the values.yaml of environment helm chart.

So, when we try to give these URL details to Gateway Server, it should be able to easily connect because my gateway server also is going to be deployed in the same Kubernetes cluster.

With this, we have successfully installed Keycloak with the helm charts.

Similarly, we need to set up other components like Kafka and Grafana, Prometheus components.

But before you try to do that, since we are going to do lot many installations into your Kubernetes

cluster, it may not work inside your local system because you will have very less memory inside your

local system.

So, to some extent we can try to overcome this challenge by changing some settings.

For the same, we can go to the Docker desktop dashboard here, I can click on these settings.

After that, I will click on these resources Under these resources, try to increase the memory that you

are trying to allocate to that Docker desktop.

So, to overcome this challenge I have increased the number of CPUs to 6 and the memory from 8GB

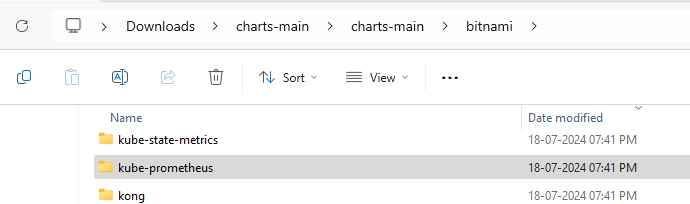
to 12GB.

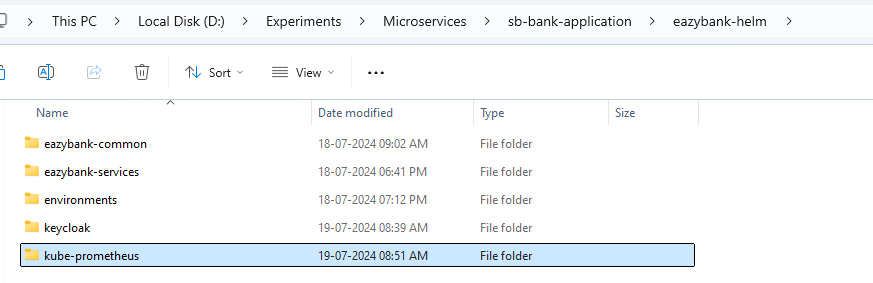
Install Kafka in Kubernetes Cluster using Helm Chart

Install Prometheus in Kubernetes Cluster using Helm Chart

Inside this lecture, let us try to set up Prometheus inside our Kubernetes cluster so that my Prometheus can scrape all the metrics from my individual microservices.

Post that we are going to set up Grafana related components and we will make our Grafana to talk with the Prometheus, to search the metrics and to set up the dashboards.



Copy this folder to   


As usual, we cannot directly install these helm charts into the Kubernetes cluster.

We need to make some changes inside the values.yaml.

So to get started with the changes, we need to open the values.yaml.

**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\kube-prometheus\values.yaml**

Here I am going to search for a setting with the name **additionalScrapeConfigs**.

So here we have a detail related to scrape configs using these configs only my Prometheus will decide to scrape the metrics from the microservices.

  additionalScrapeConfigs:

    enabled: false

    type: external

So as of now you can see the additional configs is disabled.

We need to enable this to true and post that we should change this type to internal because we do not want our Prometheus to scrape anything from outside of the Kubernetes cluster.

 additionalScrapeConfigs:

    enabled: true

    type: internal

Everything we have inside the Kubernetes cluster only so we can change this value to internal.

After making these changes, we need to provide the details of our microservices using which my Prometheus can connect with them and read the metrics of them with the help of actuator Prometheus URL.

So, for the same you can see here under the internal job list as of now it is empty.

    internal:

      jobList: []

So here I am going to mention some configurations, the same kind of configurations we also mentioned when we tried to set up Prometheus inside our local system.

So, under this job list, I am going to mention all the job names using which my Prometheus is going to fetch the metrics from the individual microservices.

    internal:

      jobList: [

        {

            "job\_name": "configserver",

            "metrics\_path": "/actuator/prometheus",

            "static\_configs": [

              {

                "targets": ["configserver:8071"]

              }

            ]

       },

       {

            "job\_name": "eurekaserver",

            "metrics\_path": "/actuator/prometheus",

            "static\_configs": [

              {

                "targets": ["eurekaserver:8070"]

              }

            ]

       },

       {

            "job\_name": "accounts",

            "metrics\_path": "/actuator/prometheus",

            "static\_configs": [

              {

                "targets": ["accounts:8080"]

              }

            ]

       },

       {

            "job\_name": "loans",

            "metrics\_path": "/actuator/prometheus",

            "static\_configs": [

              {

                "targets": ["loans:8090"]

              }

            ]

       },

       {

            "job\_name": "cards",

            "metrics\_path": "/actuator/prometheus",

            "static\_configs": [

              {

                "targets": ["cards:9000"]

              }

            ]

       },

       {

            "job\_name": "gatewayserver",

            "metrics\_path": "/actuator/prometheus",

            "static\_configs": [

              {

                "targets": ["gatewayserver:8072"]

              }

            ]

       }

      ]

So, I have mentioned all these details in a Json format.

So here it has a list of jobs and each job is separated by comma value.

If you see here first, I have mentioned what is the job name for config server the job name is going to be config server. And what is the metrics path and what is the location at which my Prometheus need to connect to get the metrics.

So, this is a service name inside our Kubernetes for config server and it is available at the port 8071.

Similarly for Eureka Server, I have mentioned accounts, loans, and cards gateway server.

So, for whatever microservices you need, you can mention those job details here.

So, once we have defined these details, my Prometheus should be able to scrape metrics from my individual microservices.

So as a next step, I can try to build the Prometheus related helm chart. For the same,

let me go inside the kube-prometheus folder and here I am going to run the command, which is

**> helm dependencies build**

PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm\kube-prometheus> **helm dependencies build**

So, this will compile my Prometheus helm chart post that we can try to install the same inside our Kubernetes cluster.

So, to install Prometheus, we need to go back and run the command, which is

**> helm install**

And what is the release name that we want to give. The release name that we can give here is Prometheus itself.

After the release name, I can mention the folder name where my helm chart is present.

It is present under the folder with the name kube-prometheus.

So let me try to execute this command.

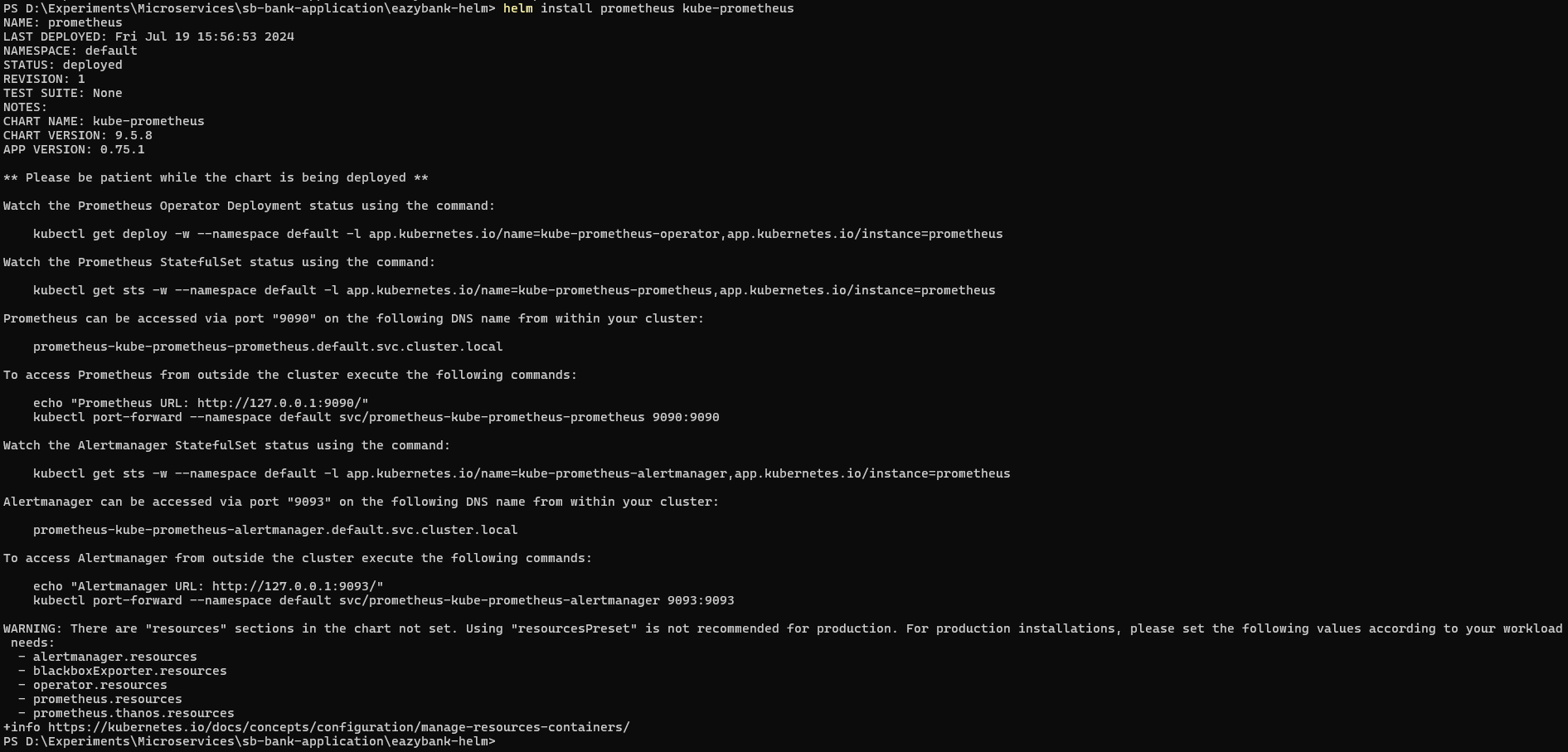
**PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm> helm install prometheus kube-prometheus**

This will do the installation of Prometheus inside my local system.

So, by default Prometheus is set up with the help of cluster IP, so we cannot really access it.

But for some reason, if you want to access the same, you can run these commands which are related

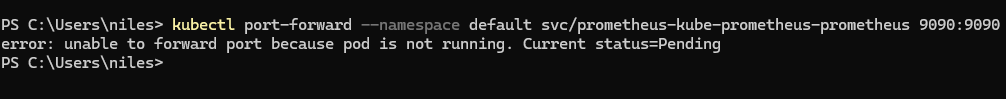
to kubectl port forward.



So, using the kubectl port forward command, we can temporarily expose a service.

So let me try to execute these commands by cleaning the console.

So, I'm trying to execute the command and this should start exposing my Prometheus but seems still my installation of Prometheus is going on.



Let us wait for few minutes.

So, I am going to wait for one two minutes behind the scenes,

I have waited for one minute.

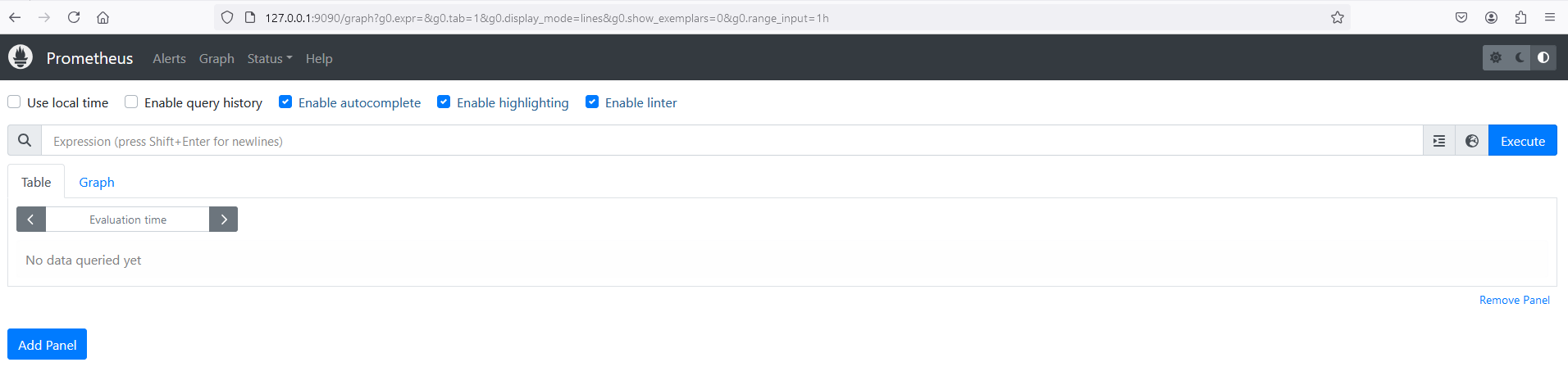
Let me try to execute this command again.

**PS C:\Users\niles> kubectl port-forward --namespace default svc/prometheus-kube-prometheus-prometheus 9090:9090**

Forwarding from 127.0.0.1:9090 -> 9090

Forwarding from [::1]:9090 -> 9090

Handling connection for 9090

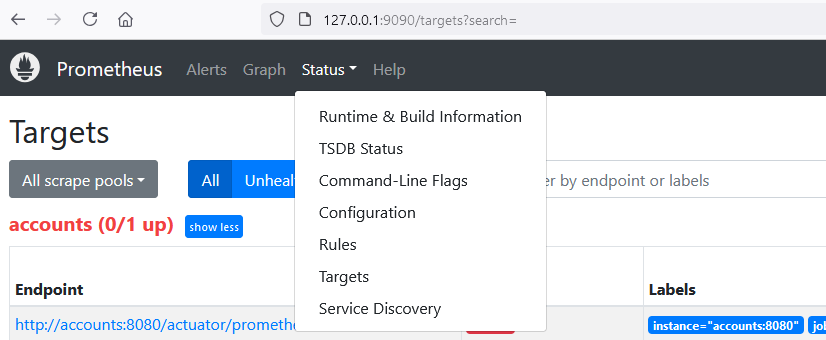


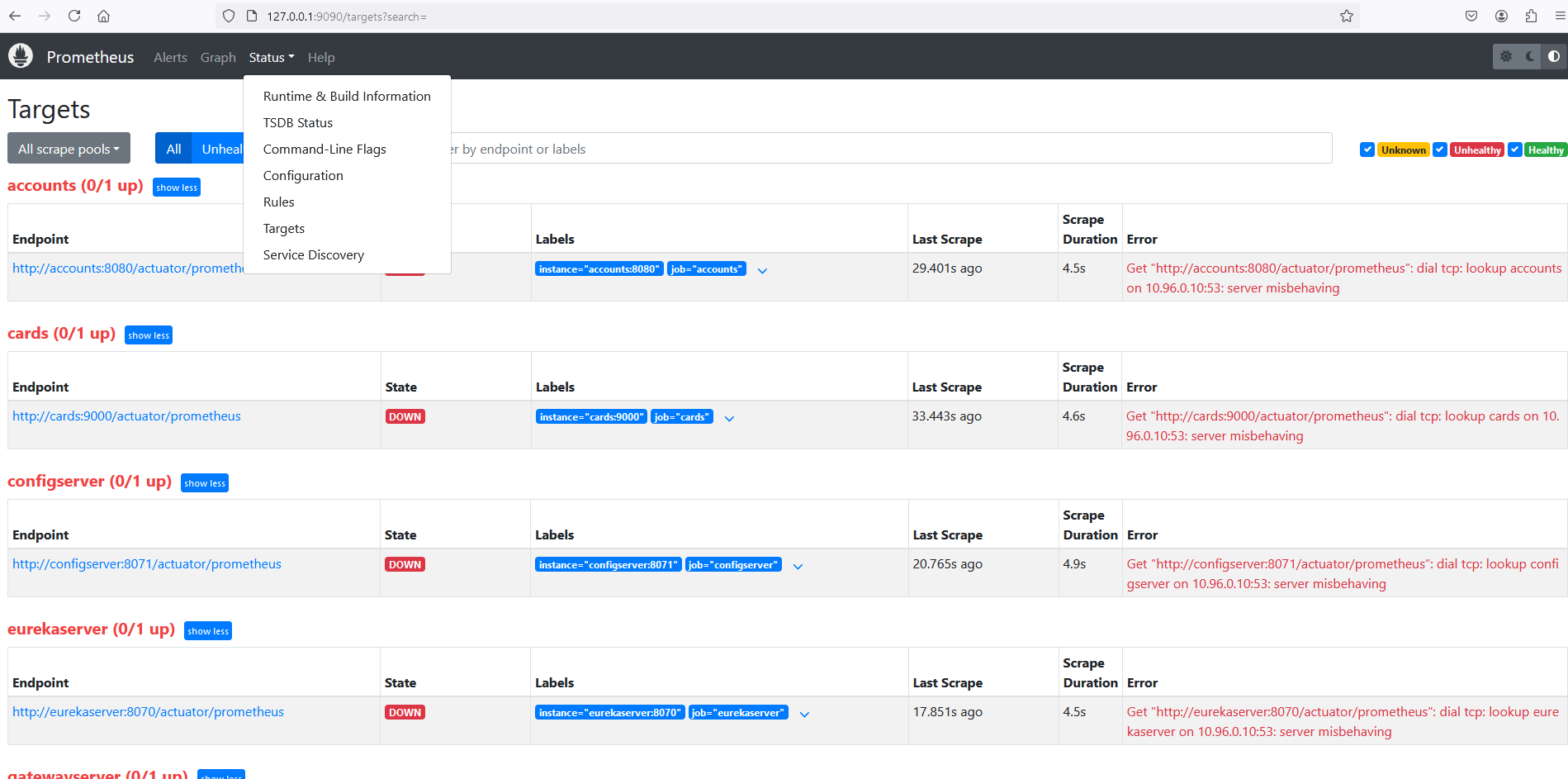
And this time you can see the forwarding of the network is successfully into our local system.

So now I should be able to access the Prometheus at the port 9090.

So here inside my browser let me open 9090 port and this is my Prometheus.

Here if I go to the targets, I should be able to see all the targets.





As of now, my accounts cards, config server, Eureka server all of them we did not setup.

That is why the status for all of them is showing as in red colour.

If you can scroll down, Prometheus will also try to monitor many other metrics inside your Kubernetes cluster.

So that is why you can see many other components inside the target's location. Since this is meant

for production ready.

Prometheus Behind the scenes, it is trying to monitor many other components inside your Kubernetes cluster, but we are more interested about our own microservices.

We can validate the same once we set up our own microservices into the Kubernetes cluster.

For now, I can stop this command by pressing Ctrl C and that will stop exposing my Prometheus into

my local network.

Now, if I try to refresh this page, I will not get any response because the Prometheus is not exposed

to the local system.

With this, we successfully set up Prometheus inside our Kubernetes cluster.

As a next step, let us try to set up Grafana related components like Loki, Tempo and Grafana.

And with that we should be done with all the required components for our microservices end to end testing.

Install Grafana-Loki and Tempo in Kubernetes Cluster using Helm Chart

As a next step, we are interested to set up Grafana related components inside our Kubernetes cluster.

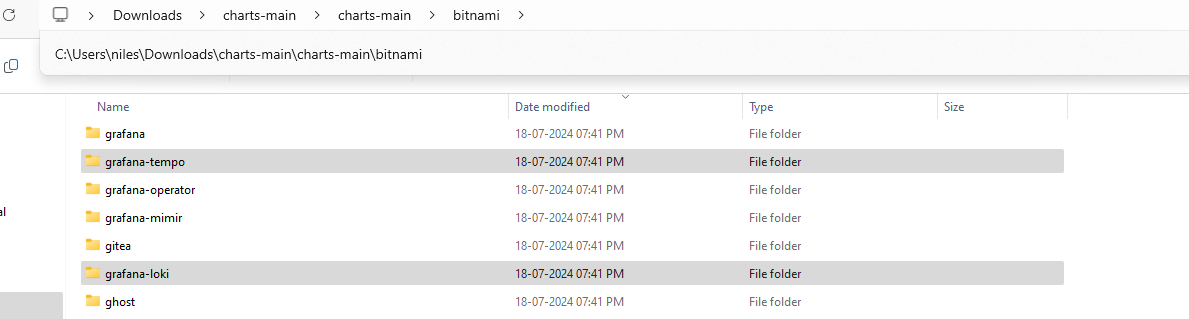
So, inside our microservice network we use Loki, Tempo and Grafana.

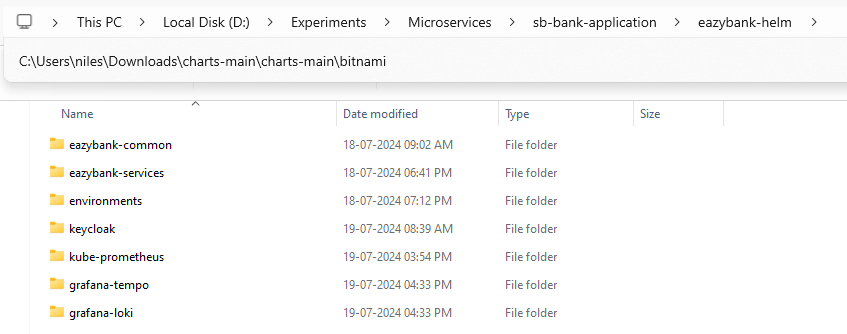
So, these three components we need to set up with the help of helm chart.

So, before we try to set up the actual Grafana, first we need to make sure we set up the Loki and

Tempo.

For the same, I am going to copy these folders of grafana-loki and grafana-tempo into the helm folder and here I am going to paste them.





First, let me try to set up grafana-loki because Loki is responsible to aggregate all the logs

that are generated by my individual microservices.

So to install Loki, we don't have to make any changes inside the values.yaml.

We should be fine with all the default values available inside this chart. So, I can straight away compile this chart and install inside my local Kubernetes cluster.

So here, let me navigate into my grafana-loki folder and post that I am going to run the command,

which are helm dependencies build.

**PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm\grafana-loki>**

**PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm\grafana-loki> helm dependencies build**

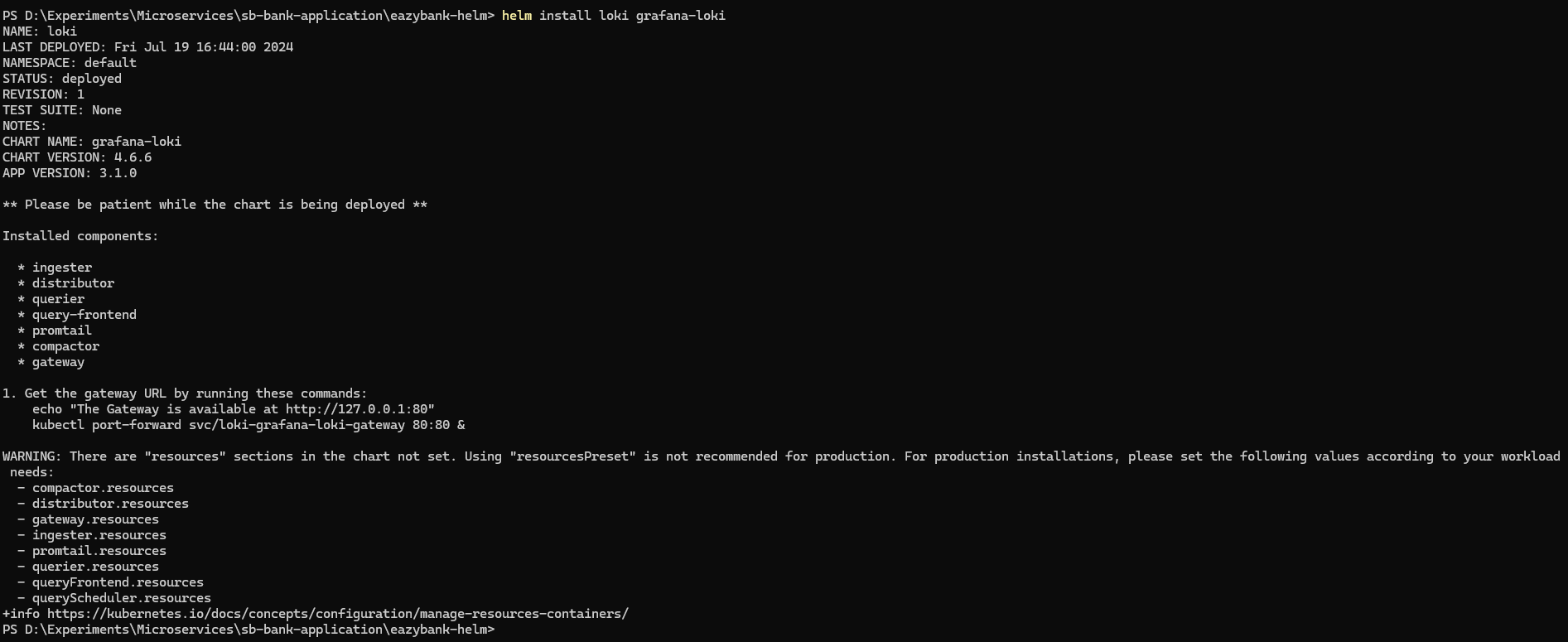
So here you can see my Loki has dependency on other charts like Memcached. That is why it is trying to download all those dependencies.

So now my compilation is successful.

I can try to install the Loki helm chart with the help of helm install command post that I am going

to give a name to my release.

**PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm> helm install loki grafana-loki**



The name is going to be Loki, but the chart name is going to be Grafana-Loki.

But since I am in the wrong location, so let me go back to the parent location and here I am going to

run the command, which is helm install and the release name is going to be Loki.

And inside this folder we have a chart with the name grafana-loki, the same I want to install inside

my Kubernetes cluster.

So, this will install many components inside my Kubernetes cluster like ingested, distributor, querier,

Promptail, compactor, gateway.

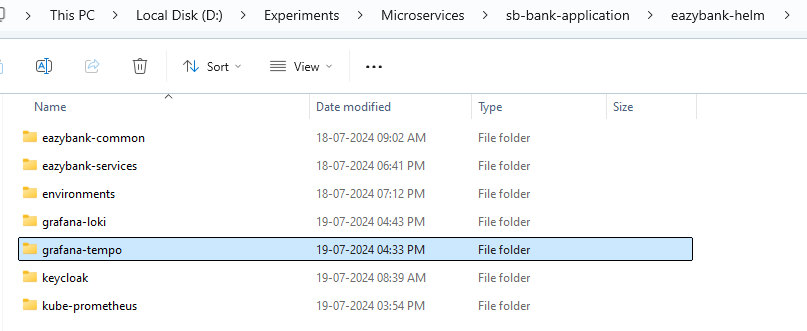
Can you imagine setting up all these without helm chart?

Your Kubernetes administrator will need a lot of help from the developers and the Grafana admin to set up these and it may need months of effort to have a proper setup, whereas with Helm it is super,

super easy.

As a next step we can set up the tempo as well. For the same, let me go to the folder location.

Here we have a chart with the name grafana-tempo.



So let me open the same inside this tempo chart, we need to change few values inside the values.yaml.

**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\grafana-tempo\values.yaml**

So let me open this.

Inside this file I am going to search for OTLP.

    otlp:

      http: false

      grpc: false

So, these are related to open telemetry configurations.

So as of now you can see the open telemetry communication with the help of Http protocol and gRPC protocol is disabled.

We need to enable both.

    otlp:

      http: true

      grpc: true

Then only the open telemetry Java agent that we have inside our individual microservices.

It can send the tracing details to the tempo.

By default, OTLP is disabled inside this helm chart.

So that is why we need to enable them by overriding these values from false to true.

So let me save this file post that I am going to compile the helm chart, which is grafana-tempo.

So here, let me execute the command, which is helm dependencies build.

**PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm\grafana-tempo> helm dependencies build**

So, this will download all the dependencies of tempo.

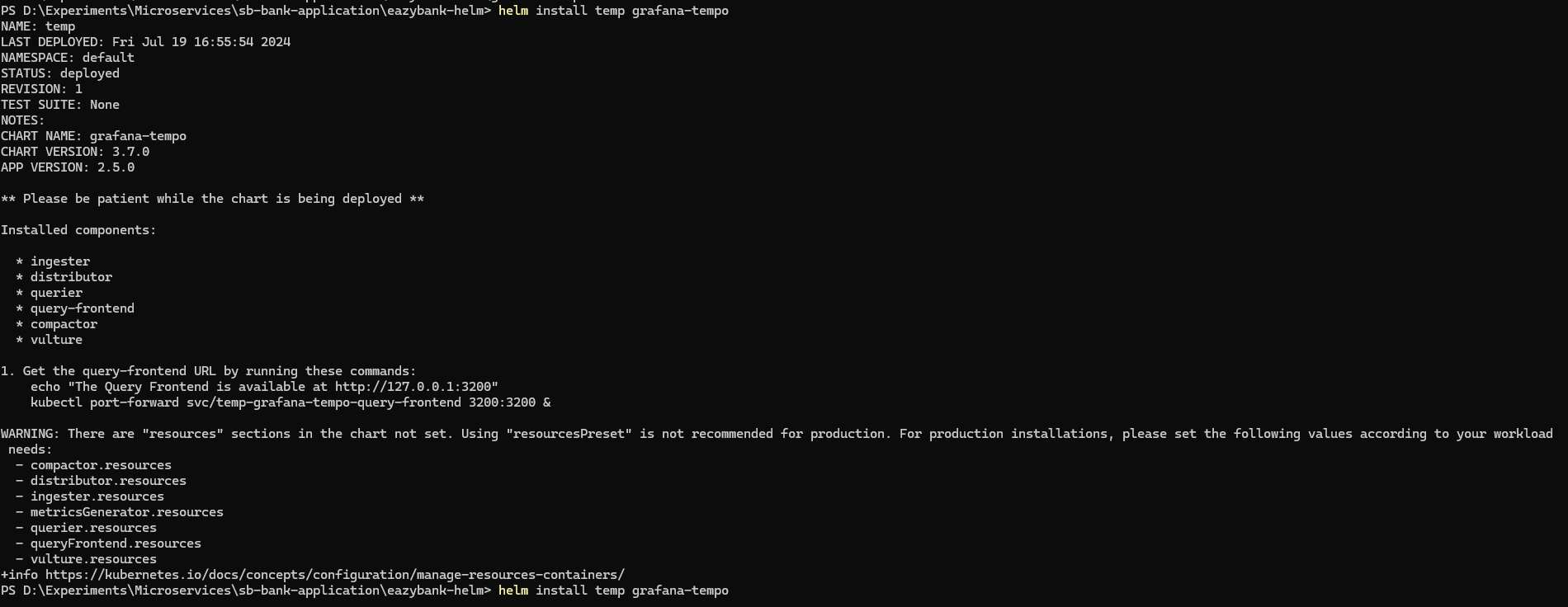
Once the build is completed, I will navigate back to the parent folder, from the parent folder,

I am going to run the command which is helm install and the name of the release I want to give here is tempo.

And post that what is a helm chart name, the helm chart name is grafana-tempo.

So let me try to execute this.

**PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm> helm install tempo grafana-tempo**



So, this will give some instructions like all these components are installed like ingested, distributor,

querier, query-frontend and compactor and vulture.

But here we have a problem, which is our individual microservices need a tempo url to which my open telemetry is going to send the details of tracing details.

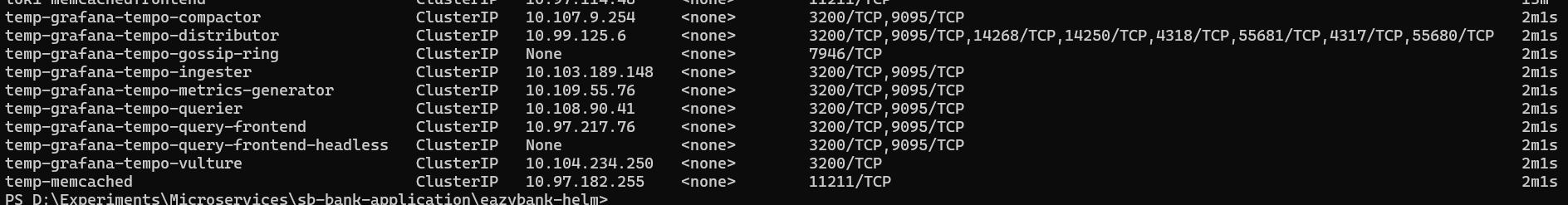
So here inside the output, we do not have the any information about the URL that we need to use.

So, to overcome this challenge, we can try to run the command, which is kubectl get services and this

command is going to list all the services installed inside your Kubernetes cluster.

So as of now, you can see we have installed so many services related to Loki, Prometheus and similarly related to keycloak, Kafka.

So, if you can scroll down, there are some information related to tempo, so there are many tempo related services.



Like there is a tempo related service with the name gossip-ring, ingester, generator, querier,

vulture, compactor, distributor.

So, the service that our open telemetry should connect is this **distributor**.



So that is why take the service name and we need to make sure we are mentioning the same inside the configmap values.

**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\environments\dev-env\values.yaml**

global:

...

  otelExporterEndPoint: http://temp-grafana-tempo-distributor:4317

...

OR

global:

...

  otelExporterEndPoint: http://tempo-grafana-tempo-distributor:4317

...

So here I have mentioned the same here and the port number must be 4317.

So, this is how we are establishing a link between our individual microservice open telemetry with the grafana-tempo.

So, you may have question like how do I know that I need to use only distributor but not the other service?

So based upon my experience and official documentation information, I can identify this information

and I am trying to spoon-feed you this information.

So, whenever you have such challenges or questions, always go to the official documentation and you should be able to get this information.

With this, we successfully installed Grafana Tempo and Loki as well as the last step we need to set

up the Grafana.

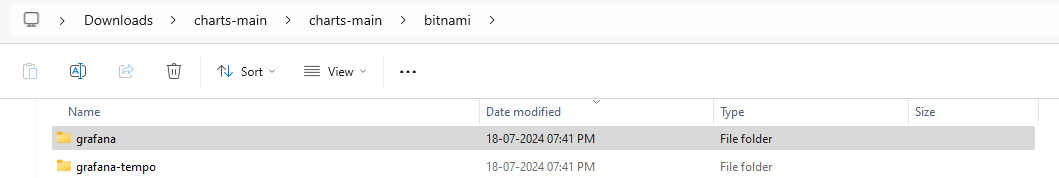
Install Grafana in Kubernetes Cluster using Helm Chart

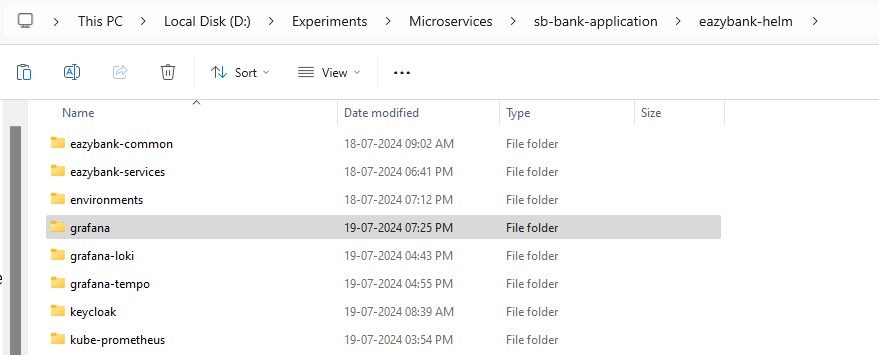
As a last step, we need to set up Grafana inside our Kubernetes cluster before we try to deploy all

our microservices.

So, to install Grafana from the Bitnami folder, I am trying to copy the Grafana related helm chart with

the name Grafana and paste the same inside the helm folder that we are mentioning.





Like I said before, I am going to check in this entire helm folder into the GitHub repo.

Please use these helm charts only that I have checked in into the GitHub repo so that you do not have to make all the changes that I have done inside the values.yaml.

So now before we try to set up this Grafana, as usual, we need to make some changes inside the values.yaml of Grafana Helm chart.

**D:\Experiments\Microservices\sb-bank-application\eazybank-helm\grafana\values.yaml**

So let me open this inside this values.yaml I need to look for the information on how to provide the

data source details of Tempo, Loki, and Prometheus.

Like we discussed in the previous sections, Grafana can connect to these components like

Prometheus, Loki, and Tempo.

We can provide these data source details to Grafana using Yaml configurations, or we can also manually set up the data sources from the UI of the Grafana.

Since we want to avoid that manual task, we need to look for an information on how to set up the datasource details inside these values.yaml.

So, for the same here I am going to search for data source.

##

datasources:

  ## @param datasources.secretName The name of an externally-managed secret containing custom datasource files.

  ##

  secretName: ""

  ## @param datasources.secretDefinition The contents of a secret defining a custom datasource file. Only used if datasources.secretName is empty or not defined.

  ## Example:

  ## secretDefinition:

  ##   apiVersion: 1

  ##   datasources:

  ##   - name: Prometheus

  ##     type: prometheus

  ##     url: http://prometheus-prometheus-server

  ##     access: proxy

  ##     isDefault: true

  ##

So here there is a section about data sources and here they have given an information saying that whenever we want to set up a data source, we need to define the data source details under the secret definition.

So here there is an element secret definition under this element secret definition, we need to provide the data source details like they have mentioned here.

They have mentioned a sample connection, details to Prometheus.

So, using the same syntax, we need to set up the data source details for Grafana.

So, under the secret definition we need to define API version data sources and what are your data sources?

So let me delete these curly braces and navigate to the next line.

Here inside this next line, I am going to mention few configurations like you can see.

So here I have mentioned API version as one and

post that I have mentioned some delete data sources if already existed in my Grafana with these names and

at last under datasources I have mentioned three data source details of Prometheus, Loki, and Tempo

datasources:

  ## @param datasources.secretName The name of an externally-managed secret containing custom datasource files.

  ##

  secretName: ""

  ## @param datasources.secretDefinition The contents of a secret defining a custom datasource file. Only used if datasources.secretName is empty or not defined.

  ## Example:

  ## secretDefinition:

  ##   apiVersion: 1

  ##   datasources:

  ##   - name: Prometheus

  ##     type: prometheus

  ##     url: http://prometheus-prometheus-server

  ##     access: proxy

  ##     isDefault: true

  ##

secretDefinition:

    apiVersion: 1

    deleteDatasources:

      - name: Prometheus

      - name: Loki

      - name: Tempo

    datasources:

      - name: Prometheus

        type: prometheus

        uid: prometheus

        url: http://prometheus-kube-prometheus-prometheus:9090

        access: proxy

        orgId: 1

        basicAuth: false

        isDefault: false

        version: 1

        editable: true

        jsonData:

          httpMethod: GET

      - name: Tempo

        type: tempo

        uid: tempo

        url: http://tempo-grafana-tempo-query-frontend:3200

        access: proxy

        orgId: 1

        basicAuth: false

        isDefault: false

        version: 1

        editable: true

        jsonData:

          httpMethod: GET

          serviceMap:

            datasourceUid: 'prometheus'

      - name: Loki

        type: loki

        uid: loki

        access: proxy

        orgId: 1

        editable: true

        url: http://loki-grafana-loki-gateway:80

        jsonData:

          httpHeaderName1: "X-Scope-OrgID"

          derivedFields:

            - datasourceUid: tempo

              matcherRegex: "\\[.+,(.+),.+\\]"

              name: TraceID

              url: '$${\_\_value.raw}'

        secureJsonData:

          httpHeaderValue1: "tenant1"

## Create notifiers from a configMap

So, this is the DNS name of Prometheus inside my Kubernetes cluster, and similarly for Tempo.

So inside Tempo we have many services like we discussed previously, so the service that we need to

use for connecting Grafana and Tempo is **tempo-grafana-tempo-query-frontend**.

So, this is the one that we need to use.

And for this the port is going to be 3200.

You can confirm the same here for the query frontend the port number is 3200.

So the same I'm trying to mention inside the values.yaml.

Now coming to the Loki. So, for Loki we need to connect to the Loki gateway service and this is the service name and it is exposed at the port 80.

And at last, since we want to integrate Loki and Tempo, I have mentioned these derived fields configurations just like how we did during our local testing.

So, once we define these values now, my Grafana should be able to connect with all the components related to Prometheus, Loki, and Tempo.

As a next step, we can compile these Grafana helm chart and install the same into our Kubernetes cluster.

So here, let me clean the console and post that I am going to navigate into the Grafana folder.

Inside this folder I am going to run the command, which is helm dependencies build.

**PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm\grafana> helm dependencies build**

Once the build is completed, we can try to navigate to the parent folder and inside this folder we

can run the command, which is helm install and the release name

**PS D:\Experiments\Microservices\sb-bank-application\eazybank-helm> helm install grafana grafana**

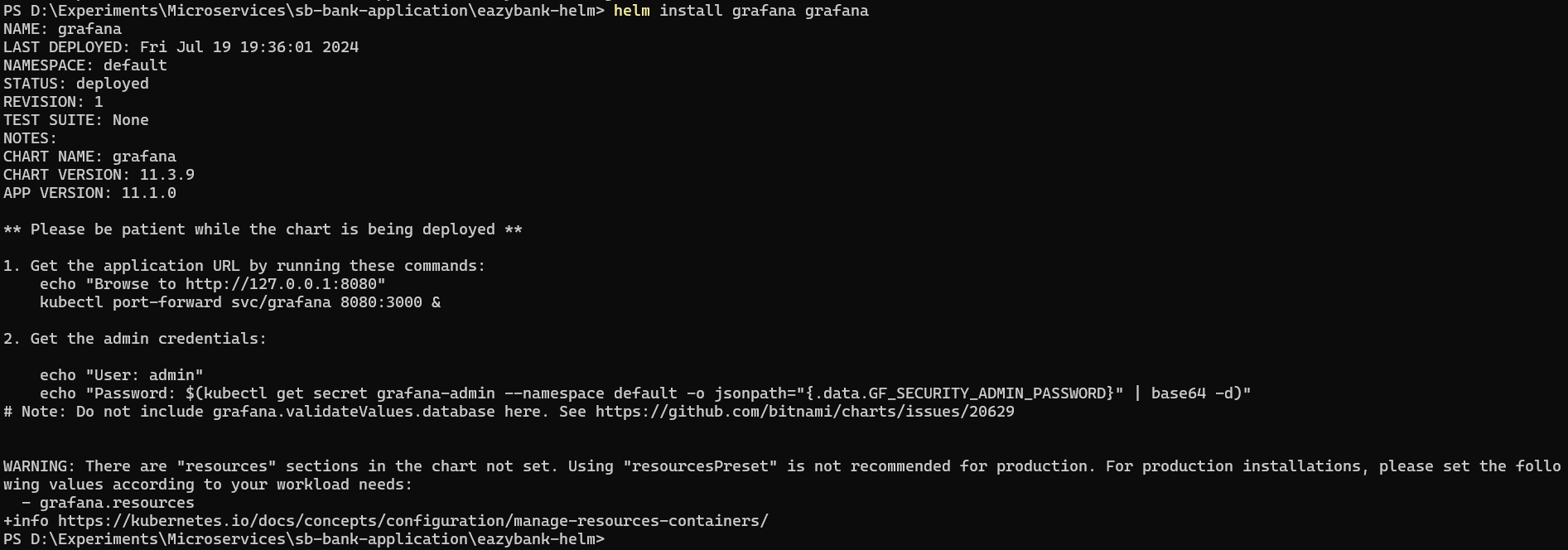
I want to give it as Grafana.

And what is the helm chart name.

The helm chart name is Grafana itself.

So, if I try to execute this command, the Grafana is going to be installed inside my local system. By

default, my Grafana is going to be exposed as a cluster IP service.



So, whenever I have some requirements to debug any issues, my Kubernetes admin can run a command which is related to kubectl port forward.

With this, my Grafana is going to be temporarily accessible with the help of these 3000-port number.

So let me try to execute this command.

So here I am executing the command.

**PS C:\Users\niles> kubectl port-forward svc/grafana 8080:3000**

**Forwarding from 127.0.0.1:8080 -> 3000**

**Forwarding from [::1]:8080 -> 3000**

**Handling connection for 8080**

**Handling connection for 8080**

After executing this command, we should be able to access the Grafana at the port 8080. Because we expose that traffic at the port 8080.

So Grafana internally is going to start at the port 3000. But here the traffic is going to be exposed at the port 8080. But this is going to create some problems because for some reason, if you start your accounts microservice as a LoadBalancer service, it is also going to expose the traffic at the port 8080.

With that reason, you may face some issues.

That is why I can kill this command and try to run the same command again.

But this time I will change this port number from 8080 to 3000 itself. This way my Grafana is going to be available at the port 3000 itself. So let me execute this command.

**PS C:\Users\niles> kubectl port-forward svc/grafana 3000:3000**

**Forwarding from 127.0.0.1:3000 -> 3000**

**Forwarding from [::1]:3000 -> 3000**

Now I will go to the browser and try to access the port 3000.

Here I am trying to access the port 3000 and you can see I got a welcome message where I can enter

username and password. But I do not know what is the username and password. For the same,

if you closely observe the instructions, there are instructions on how to get the username and password.

The username is admin, but to know the password we need to execute these two commands.

So let me go to the other terminal.

So here I am going to run the command and the username is going to be admin and this is the password. So let me copy this value.

niles@Nilesh MINGW64 ~

$ echo "Password: $(kubectl get secret grafana-admin --namespace default -o jsonpath="{.data.GF\_SECURITY\_ADMIN\_PASSWORD}" | base64 -d)"

Password: OQrkPt19Nl

Here I am going to enter the credentials and post that

I will click on the login and you can see right now I am logged in into the Grafana successfully.

I can also validate if the data source connection is successful or not.

By going to these explore under this dropdown, you can see I am able to see three data sources related

to Loki, Prometheus, and Tempo.

This confirms our Grafana setup also is completed.

As a next step, we can finally install all our microservices into the Kubernetes cluster.

Before that, let me stop this port forwarding and with this I will not be able to access my Grafana

in future

whenever I need, I can try to run the port forward command.

So let me clean the console and try to run the command, which is helm ls.

This will show you all the releases or all the installations or deployments that we have done with the

help of Helm.

So as of now we have installed Grafana, Kafka, Keycloak, Loki, Prometheus, and Tempo. And these

are the chart details that we have used to set up these components inside the Kubernetes cluster.

I hope this is all clear to you in the next lecture,

let us try to deploy our microservices into the Kubernetes cluster.

Thank you, and I'll catch you in the next lecture bye.

Install **eazybank** microservices in Kubernetes Cluster using Helm Chart

Demo of helm upgrade command

Demo of helm history and rollback commands

Demo of helm uninstall command

Quick revision of important helm commands