Introduction to Cloud Computing – Exercise 10

Scope: Helm charts

Introduction:

The aim of the lab is to get acquainted with the Helm package manager for Kubernetes. **Before performing the laboratory, restart the minicube.**

1. Create a new *chart*

The best way to get started with a new *chart* is to use the *helm* create command to create a skeleton of an example that we can rely on. Use this command to create a new *chart* named mychart in the new directory:

```
Helm create mychart
```

Helm will create a new directory in your project called mychart with the structure shown below. Let's review the structure of the created *greyhound* to find out how it works.

```
mychart
|-- Chart.yaml
|-- charts
|-- templates
|   |-- NOTEBOOK.txt
|   |-- _helpers.tpl
|   |-- deployment.yaml
|   |-- hpa.yaml
|   |-- ingress.yaml
|   |-- serviceaccount.yaml
|   '-- service.yaml
'-- values.yaml
```

Templates

The most important piece of the puzzle is the templates/ directory. This is where Helm finds definitions of YAML for our services, deployments and other Kubernetes objects. If you already have definitions of your application, all you need to do is replace the generated YAML files with your own. What you get is a working *greyhound* that can be deployed with *the helm install* command.

Note, however, that the directory is called templates, and Helm runs every file in that directory through the Go interpretation engine. Helm extends the scripting language by adding a number of utility features to *greyhound* writing. Open the service.yaml file to see what it looks like:

```
apiVersion: v1
kind: Service
metadata:
name: {{ include "mychart. fullname" .  }}
Tags:
    {{- include "mychart.labels" . | nindent 4 }}
Spec:
type: {{ . Values.service.type }}
AfterRTS:
- port: {{ . Values.service.port }}
    targetPort: http
    protocol: TCP
    name: http
Selector:
    app: {{- include "mychart.selectorLabels" . | nindent 4 } }
```

This is the basic definition *of a greyhound* service. During deployment, Helm will generate a definition that looks more like a valid service. We can perform a trial installation and enable debugging to checkthe generated definitions:

```
helm install --dry-run –debug mychart  ./mychart
...
# Source: mychart/templates/service.yaml
apiVersion: v1
kind: Service
metadata:
  Name: Mychart
  Tags:
    helm.sh/chart: Mychart-0.1.0
```

```
    app.kubernetes.io/name: mychart
    app.kubernetes.io/instance: mychart
    app.kubernetes.io/version: "1.16.0"
    app.kubernetes.io/managed-by: Helm
Spec:
  type: ClusterIP
  Ports:
    - port: 80
      targetPort: http
      protocol: TCP
      name: http
  Selector:
    app.kubernetes.io/name: mychart
    app.kubernetes.io/instance: mychart
...
```

Values

The template in service.yaml uses Helm-specific objects: Chart and . Values. The former provides metadata about *the greyhound* to definitions such as name or version. The latter object . Values is a key element of Helm charts, used to reveal the configuration that can be set during deployment. The default values for this object are defined in the values.yaml file. Try changing the default value of service. Port and perform another test installation. You should notice that the port in the service and containerPort in the deploymentwill change. The service.port value is used here to ensure that service objects and deployment work together correctly. Using templates can significantly reduce schemas and simplify definitions.

If you want to change the default configuration, you can make overrides directly on the command line:

```
helm install --dry-run --debug mychart ./mychart --set service.port=8080
```

For a more advanced configuration, we can create a YAML file containing overrides using the --*values* option.

Helpers and other features

The service.yaml template also uses the parts defined in _helpers.tpl, as well as functions such as replace. The Helm documentation provides a deeper guide to helpers, explaining how functions, parts, and flow control can be used when creating a chart.

Documentation

Another useful file in the templates/ directory is the NOTES.txt file. This is a template text file that is added after a successful *greyhound* deployment. As we will see, when we implement our first greyhound, it is a useful place to briefly describe the next steps regarding the useof e created *greyhound*. Because the NOTES file.txt is triggered by a scripting language interpreter, you can use the syntax to write out draft commands to obtain an IP address or password from a Secret object.

Metadata

As mentioned earlier, the Helm chart consists of metadata that helps describe the application, define restrictions on the minimum version required for Kubernetes and/or Helm, and manage the *chart* version. All of this metadata is in the Chart.yaml file. The Helm documentation describes the various fields of this file.

2. Implementation of your own *greyhound*

The greyhound generated in the previous step is configured to run the NGINX server provided through Kubernetes. By default , a service of type ClusterIP will be created, so NGINX will only be exposed internally in the cluster. To access it from the outside, we will use NodePort instead. We can also set the name of the Helm installation so that we can easily refer to it. Let's go ahead and deploy our NGINX Helm *chart* using the helm install command:

```
helm install example ./mychart
NAME: example
LAST DEPLOYED: Tue May 31 23:24:22 2022
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTEBOOK:
1. Get the application URL by running these commands:
  export NODE_PORT=$(kubectl get --namespace default -o
jsonpath="{.spec.ports[0].nodePort}" services example-mychart)
  export NODE_IP=$(kubectl get nodes --namespace default -o
jsonpath="{.items[0].status.addresses[0].address}")
  echo http://$NODE_IP:$NODE_PORT
```

The Helm install output displays a useful summary of the installation status and a rendered NOTES.txt file to explain what's next. Run commands in the output to get the URL and to access and run NGINX in a browser.

If all went well, you should see the NGINX welcome page as shown above.

3. Modification *of the greyhound* to implement your own service

The generated *chart* creates a Deployment object marked to run the *greyhound* provided by the default values. This means that all we have to do to startanother ę service is to change the image in the values.yaml file.

We're going to update the *chart* to launch the to-do list app available in the Docker Hub. In values.yaml, update the image variables to refer to both "todo":

```
Image:
Repository: prydonius/todo
Tag: 1.0.0
pullPolicy: IfNotPresent
```

When working on editing a *chart,* it is a good idea to pass it through the *lint* tool to make sure that you follow the requirements of the yaml files and that our file has no error. Run the helm lint command to see the linter in action:

```
Helm Lint ./Mychart
==> Linting./mychart
```

```
[INFO] Chart.yaml: icon is recommended

1 chart(s) linted, no failures
```

Lint hasn't identified any majorproblems withthe variables, so we can move on. (NOTE, WE DO NOT DO THIS) However, as an example, here's what *the lint* tool can display if it found errors:

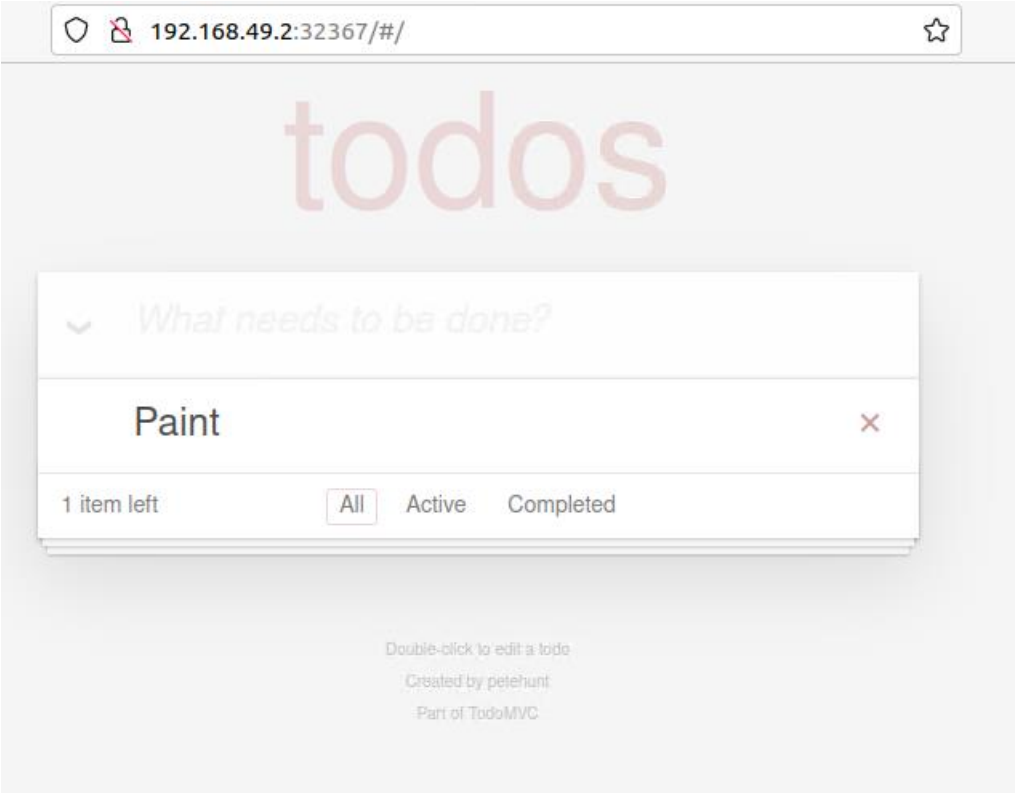
```
echo "malformed" > mychart/values.yaml
Helm Lint ./Mychart
==> Linting./mychart
[INFO] Chart.yaml: icon is recommended
[ERROR] values.yaml: unable to parse YAML: error unmarshaling JSON: while decoding
JSON: json: cannot unmarshal string into Go value of type chartutil. Values
[ERROR] templates/: cannot load values.yaml: error unmarshaling JSON: while decoding
JSON: json: cannot unmarshal string into Go value of type map[string]interface {}
[ERROR]: unable to load chart
      cannot load values.yaml: error unmarshaling JSON: while decoding JSON: json:
cannot unmarshal string into Go value of type map[string]interface {}

Error: 1 chart(s) linted, 1 chart(s) failed
```

This time, *lint* informs us that it was unable to properly analyze the values.yaml file.

Now that the *chart* is configured correctly, restart the Helm installation to deploy the applicationwith the "todo" image:

```
helm install example2 ./mychart --set service.type=NodePort
NAME: example2
LAST DEPLOYED: Tue May 31 23:32:53 2022
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTEBOOK:
1. Get the application URL by running these commands:
export NODE_PORT=$(kubectl get --namespace default -o jsonpath="{. spec.ports[0].
nodePort}" services example2-mychart)
export NODE_IP=$(kubectl get nodes --namespace default -o jsonpath="{.items[0].
status.addresses[0].address}")
  echo http://$NODE_IP:$NODE_PORT
```

Once again, we can run commands in NOTES to get the URL to access our application.



4. Create a package

So far, in this lab, we have used the helm install command to install a local, unpacked *greyhound*. However, if you want to share your *greyhounds*(with the team or the community), you can use the Helm package to create a tar package:

```
Helm package ./mychart
```

Helm will create a package mychart-0.1.0.tgz in our working directory, using the name and version from the metadata defined in the Chart.yaml file. We can now install from this package instead of from the local directory, passing the package as a parameter to the Helm installation.

```
helm install example3 mychart-0.1.0.tgz --set service.type=NodePort
```

Dependencies

As the complexity ofthese applications in which we have many *greyhounds increases*, we may find that we will have to download various dependencies, such as a database. Helm allows you

to specify the *child*charts that will be created within the same version. Tode-finish a dependency, add the following to the Charts.yaml file:

```
dependencies:
- Name: MySQL
  Version: 8.8.6
  Repository: "https://charts.bitnami.com/bitnami"
```

Like a runtime language dependency file (such as Requirements.txt in Python), the Requirements.yaml file allows you to manage *greyhound* dependencies and their versions. When you update a dependency, a file is generated so that later the dependency download uses a known, working version. Run the following command to retrieve the MySQL dependency we defined:

```
Helm Dependency Mychart Update
Getting updates for unmanaged Helm repositories...
... Successfully got an update from the "https://charts.bitnami.com/ bitnami" chart
repository
Saving 1 charts
Downloading mysql from repo https://charts.bitnami.com/bitnami
Deleting outdated charts
$ ls ./mychart/charts
mysql-8.8.6.tgz
```

Helm found a matching version in the bitnami repository and downloaded it to the subdirectory of our *greyhound*. Now that we go and install *the chart*, we will see that MySQL objects are also created:

```
helm install example5 ./mychart --set service.type=NodePort
NAME: example5
LAST DEPLOYED: Tue May 31 23:51:56 2022
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTEBOOK:
1. Get the application URL by running these commands:
  export NODE_PORT=$(kubectl get --namespace default -o
jsonpath="{.spec.ports[0].nodePort}" services example5-mychart)
```

```
  export NODE_IP=$(kubectl get nodes --namespace default -o
jsonpath="{.items[0].status.addresses[0].address}")
  echo http://$NODE_IP:$NODE_PORT
```

You can now view a list of all running helm *greyhounds*

```
Helmet ls
```

And see what pods are created in Kubernetes

```
Kubectl get pods
```