HELM

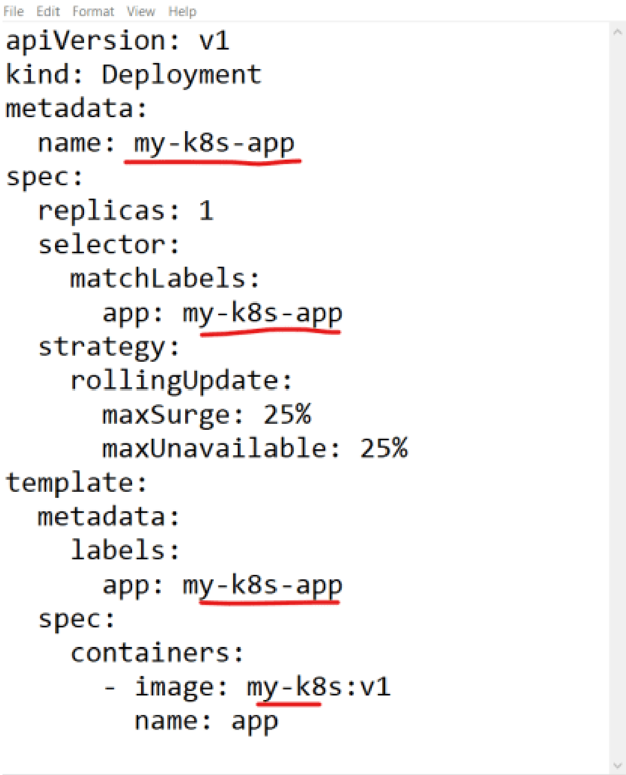
What is Helm?

Package Manager for Kubernetes

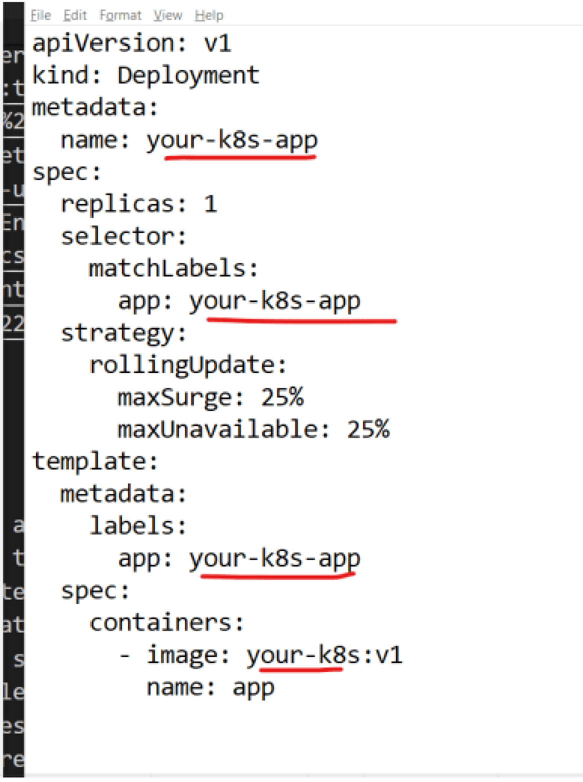
To package YAML Files

and distribute them in public and private repositories

- In order to deploy an application on k8s we need to interact with k8s API to create resources, kubectl is the tool we use to do this
- We express Kubernetes resources in YAML Files
- These files are static in nature.
- Resource files are static:
  - This is the challenge that primarily effects the declarative configuration style of applying YAML resources
  - K8s YAML files are not designed to be parametrized
  - Consider the below two manifests written to deploy two different applications



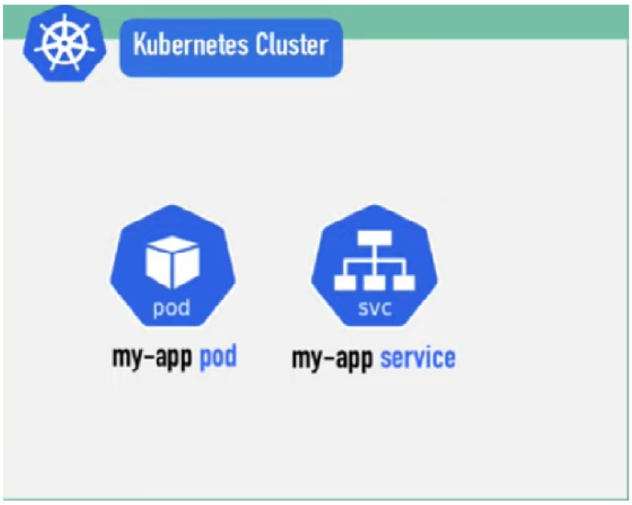
```yaml
apiVersion: v1
kind: Deployment
metadata:
  name: my-k8s-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-k8s-app
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
template:
  metadata:
    labels:
      app: my-k8s-app
  spec:
    containers:
      - image: my-k8s:v1
        name: app
```



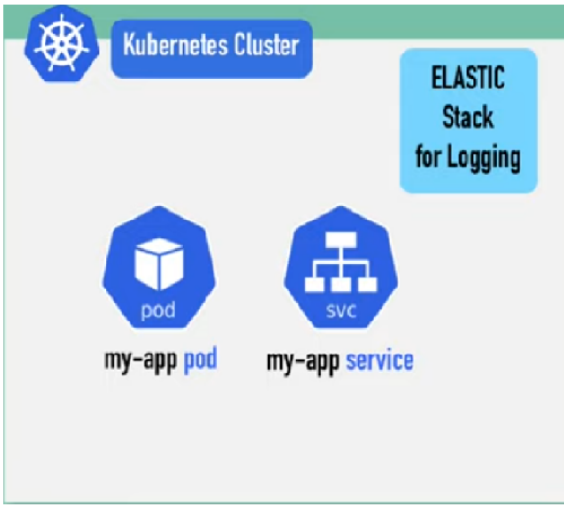
```yaml
apiVersion: v1
kind: Deployment
metadata:
  name: your-k8s-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: your-k8s-app
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
template:
  metadata:
    labels:
      app: your-k8s-app
  spec:
    containers:
      - image: your-k8s:v1
        name: app
```

  - In the above image each file is almost exactly the same, but we still cannot parametrize
- Helm to the rescue: Helm is an opensource tool used for packaging and deploying applications on k8s. It is often referred as Kubernetes Package Manager.

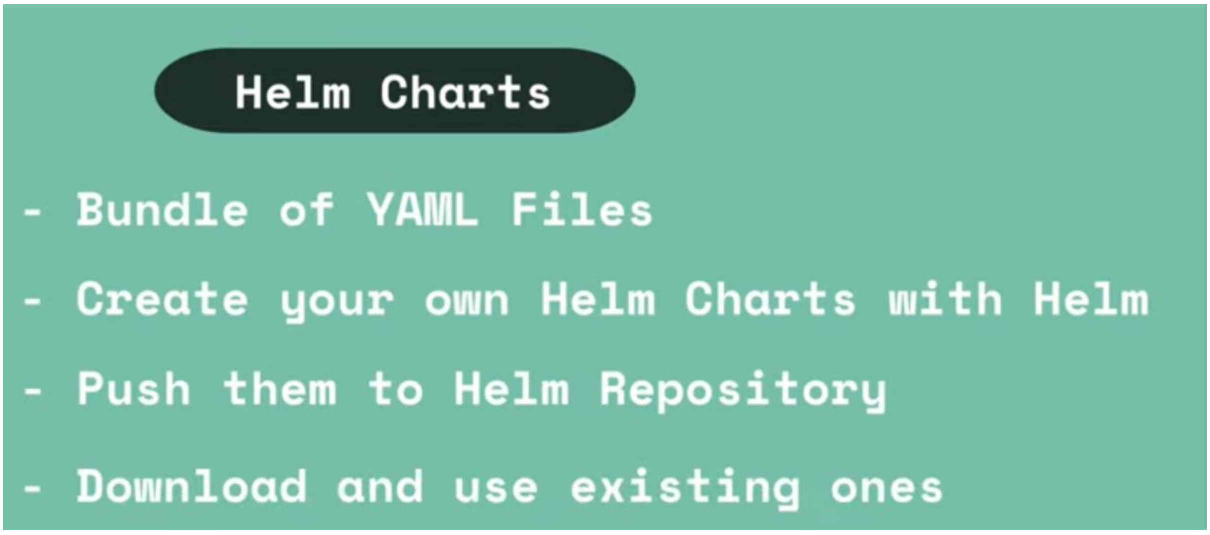
- Suppose you have a Kubernetes cluster, and you want to install ELK as a side car container, then you need to write stateful set, PV, PVC, services, config map and secrets.



Before

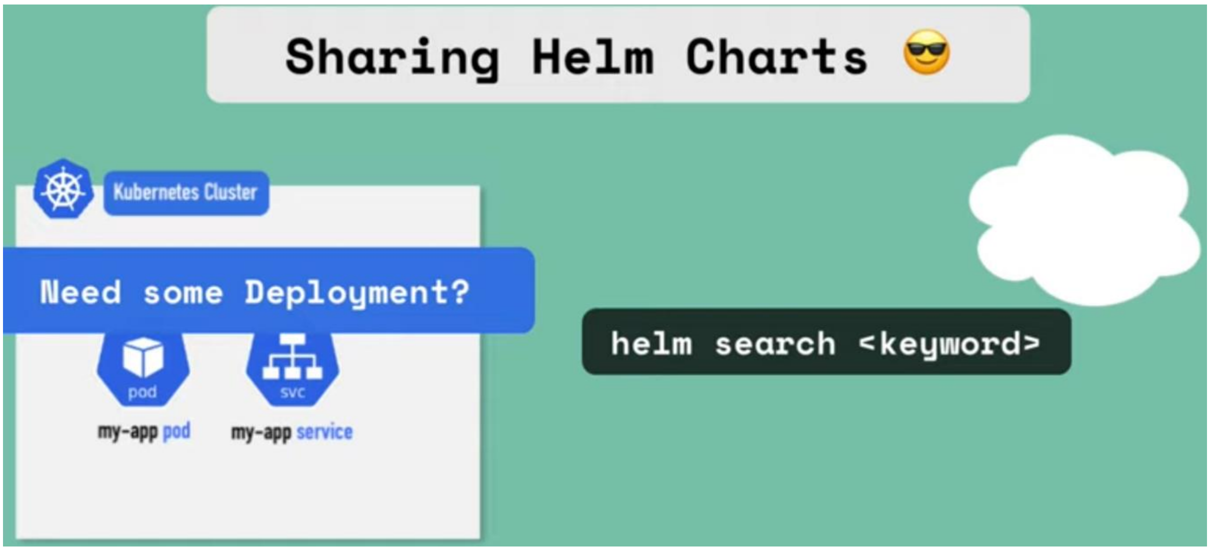


After



Helm Charts

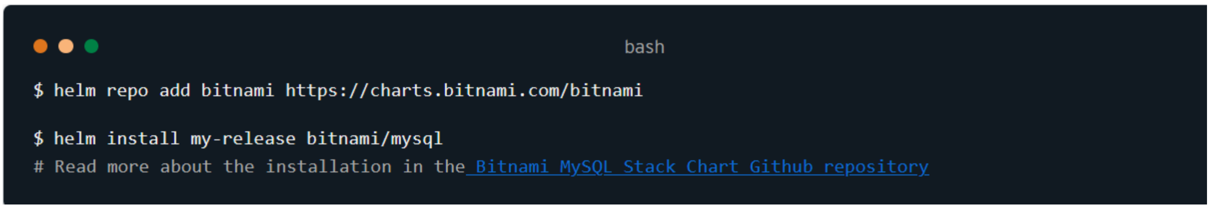
- Bundle of YAML Files
- Create your own Helm Charts with Helm
- Push them to Helm Repository
- Download and use existing ones



Sharing Helm Charts 😎

```
helm search <keyword>
```

- Helm was designed to provide an experience similar to that of package manager (apt, yum, dnf etc)
- APT operates on Debian packages and yum/dnf operates on RPM package.
- Helm operates on Charts.
- A Helm Chart contains declarative k8s resource files required to deploy an application
- Helm relies on repositories to provide access to charts
- Chart developers create declarative YAML files, package them into charts and publish them to chart repository
- End users then Helm to search for existing chart to deploy some app on to k8s
- Refer below to view a sample usage of helm chart which installs mysql.
- Link: https://bitnami.com/stack/mysql/helm



```
bash
$ helm repo add bitnami https://charts.bitnami.com/bitnami

$ helm install my-release bitnami/mysql
# Read more about the installation in the Bitnami MySQL Stack Chart Github repository
```

Let's try to understand Helm's subcommands

DNF Subcommands	Helm Subcommands	Purpose
install	install	Install an application and its dependencies
upgrade	upgrade	Upgrades an application to newer version
downgrade	rollback	Reverts the application to previous version
remove	uninstall	Delete an application

- The abstracted complexity of k8s resources:
    - Let's assume a developer has been given a task of deploying a MySQL database onto k8s.
    - Developer needs to create resources required to create containers, network and storage
    - With Helm, developer tasked with deploying a mysql database could simply search for MySQL Chart in chart repositories
- Automated Life cycle Hooks:
    - Helm provides the ability to define the life cycle hooks. Lifecycle hooks are actions that take place automatically at different stages of an application's life cycle.
    - Examples:
        - Perform a data backup on an upgrade
        - Restore data on rollback
        - Validate k8s environment prior to installation

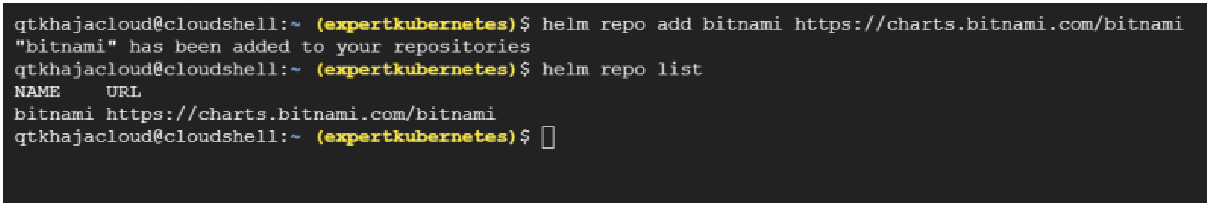
Installing Helm Link: https://helm.sh/docs/intro/install/

```
$ curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
$ chmod 700 get_helm.sh
$ ./get_helm.sh
```

Configuring Helm

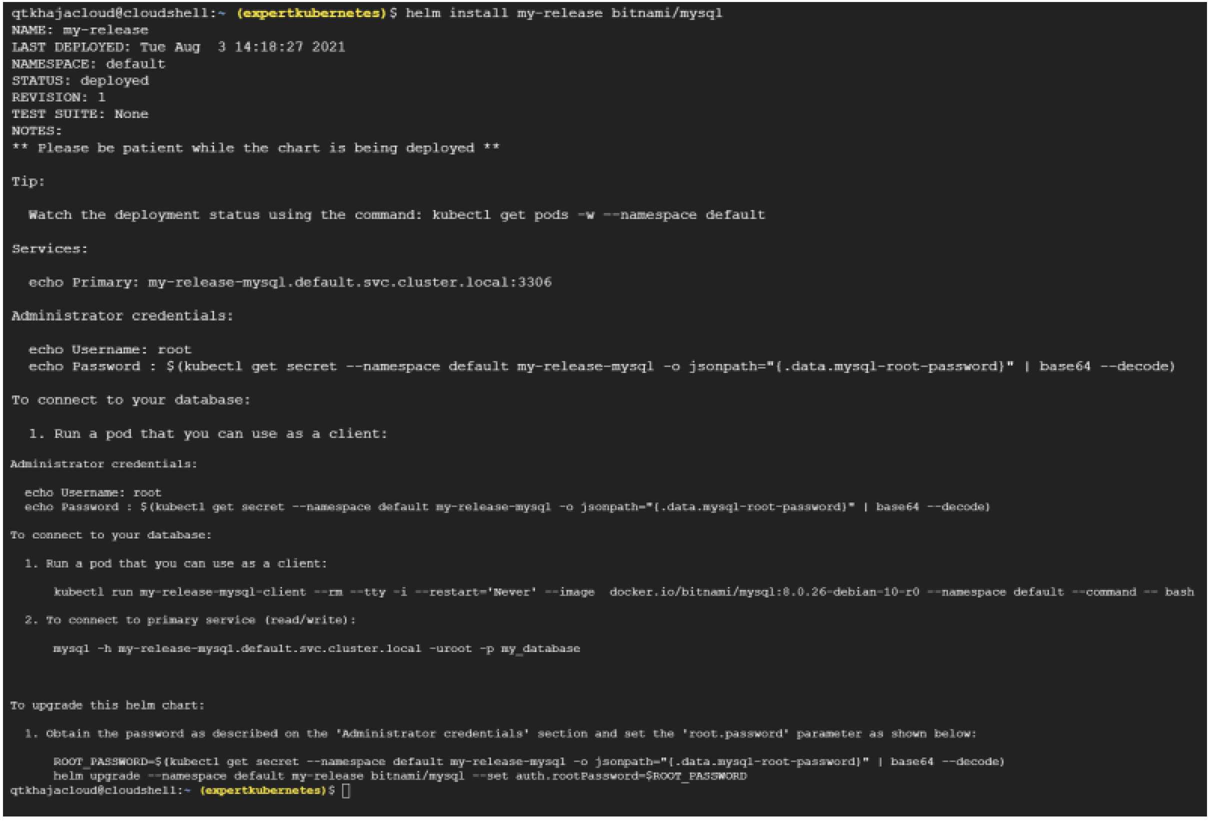
- Helm is a tool with sensible default settings that allow users to be productive without needing to perform a large of post-installation tasks
- With that being said there are several options users can change or enable to modify the Helm's behaviour.
- Adding upstream repositories:
    - Helm provides the **repo** subcommand to allow users to manage configured chart repositories. This subcommand contains additional subcommands
        - **add**: To add a chart repository
        - **list**: To list chart repositories
        - **remove**: To remove the chart repository
        - **update**: To update information on available charts locally from chart repositories
        - **index**: To generate and index file given a directory containing packaged charts
- Example: Lets install MySQL from bitnami repository

Add bitnami repository as upstream



```
qtkhajacloud@cloudshell:~ (expertkubernetes)$ helm repo add bitnami https://charts.bitnami.com/bitnami
"bitnami" has been added to your repositories
qtkhajacloud@cloudshell:~ (expertkubernetes)$ helm repo list
NAME    URL
bitnami https://charts.bitnami.com/bitnami
qtkhajacloud@cloudshell:~ (expertkubernetes)$
```

Install MySQL

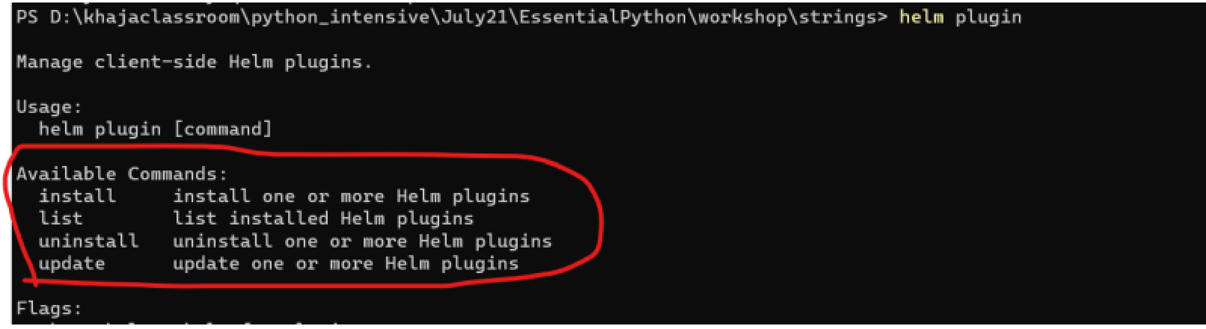


```
qtkhajacloud@cloudshell:~ (expertkubernetes)$ helm install my-release bitnami/mysql
NAME: my-release
LAST DEPLOYED: Tue Aug  3 14:18:27 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
** Please be patient while the chart is being deployed **

Tip:

  Watch the deployment status using the command: kubectl get pods -w --namespace default

Services:

  echo Primary: my-release-mysql.default.svc.cluster.local:3306

Administrator credentials:

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

To connect to your database:

  1. Run a pod that you can use as a client:

Administrator credentials:

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

To connect to your database:

  1. Run a pod that you can use as a client:

      kubectl run my-release-mysql-client --rm --tty -i --restart='Never' --image  docker.io/bitnami/mysql:8.0.26-debian-10-r0 --namespace default --command -- bash

  2. To connect to primary service (read/write):

      mysql -h my-release-mysql.default.svc.cluster.local -uroot -p my_database


To upgrade this helm chart:

  1. Obtain the password as described on the 'Administrator credentials' section and set the 'root.password' parameter as shown below:

      ROOT_PASSWORD=$(kubectl get secret --namespace default my-release-mysql -o jsonpath="{.data.mysql-root-password}" | base64 --decode)
      helm upgrade --namespace default my-release bitnami/mysql --set auth.rootPassword=$ROOT_PASSWORD
qtkhajacloud@cloudshell:~ (expertkubernetes)$
```

To uninstall MySQL helm uninstall my-release

Adding plugins:

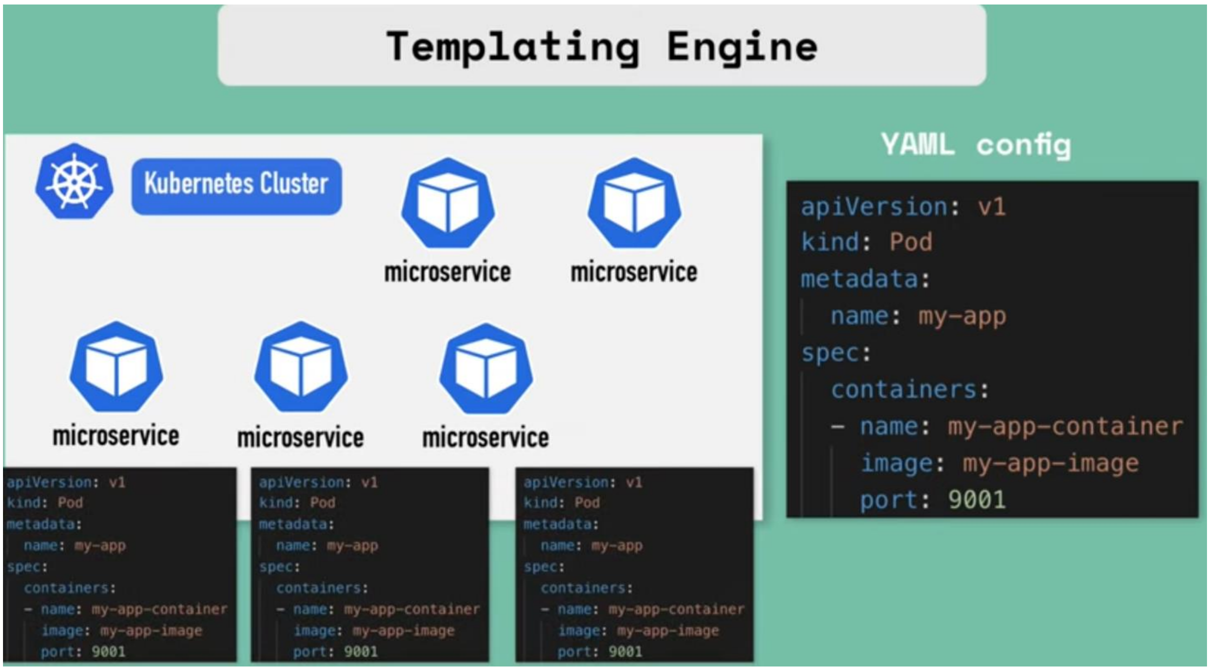
- Plugins are add-on capabilities that can be used to provide additional features to helm.
- For managing plugins, helm has a subcommand **plugin**



```
PS D:\khajaclassroom\python_intensive\July21\EssentialPython\workshop\strings> helm plugin

Manage client-side Helm plugins.

Usage:
  helm plugin [command]

Available Commands:
  install     install one or more Helm plugins
  list        list installed Helm plugins
  uninstall   uninstall one or more Helm plugins
  update      update one or more Helm plugins

Flags:
```

- Refer https://helm.sh/docs/community/related/ for some plugins

- ENVIRONMENT Variables: Helm relies on the existence of externalized environmental variables to configure low-level options
    - XDG_CACHE_HOME: Sets an alternative location for storing cached files
    - XDG_CONFIG_HOME: Sets an alternative location for storing helm configuration
    - XDG_DATA_HOME: Sets an alternative location for storing Helm Data
    - HELM_DRIVER: Sets the backend storage driver
    - HELM_NO_PLUGINS: Disables the plugins
    - KUBECONFIG: Sets an alternative Kubernetes configuration file
- Link- https://helm.sh/docs/helm/helm/
- Helm has the following paths
    - Windows:
        - Cache Path: %TEMP%\helm
        - Configuration Path: %APPDATA%\helm
        - Data Path: %APPDATA%\helm
    - Linux:
        - Cache Path: $HOME/.cache/helm
        - Configuration Path: $HOME/.config/helm
        - Data Path: $HOME/.local/share/helm

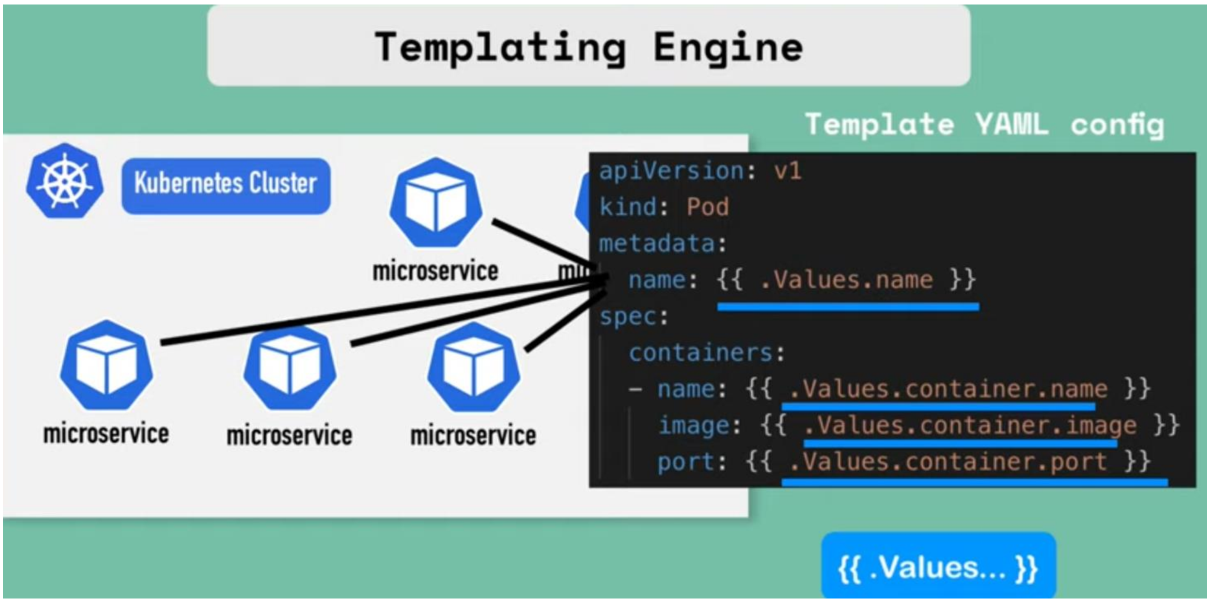


```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
  - name: my-app-container
    image: my-app-image
    port: 9001
```

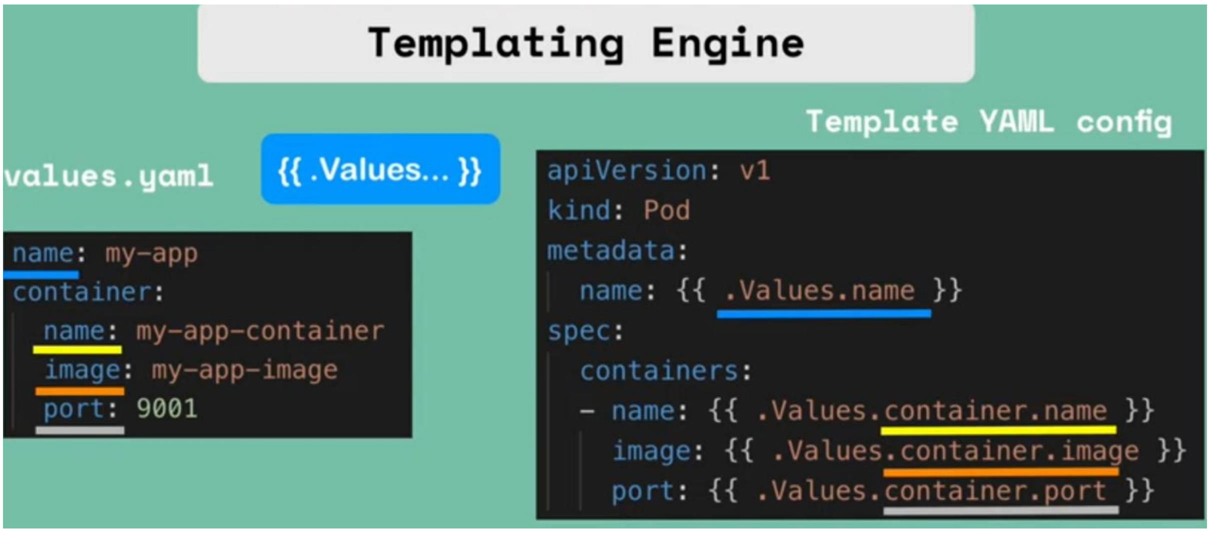
```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
  - name: my-app-container
    image: my-app-image
    port: 9001
```

```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
  - name: my-app-container
    image: my-app-image
    port: 9001
```

```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
  - name: my-app-container
    image: my-app-image
    port: 9001
```

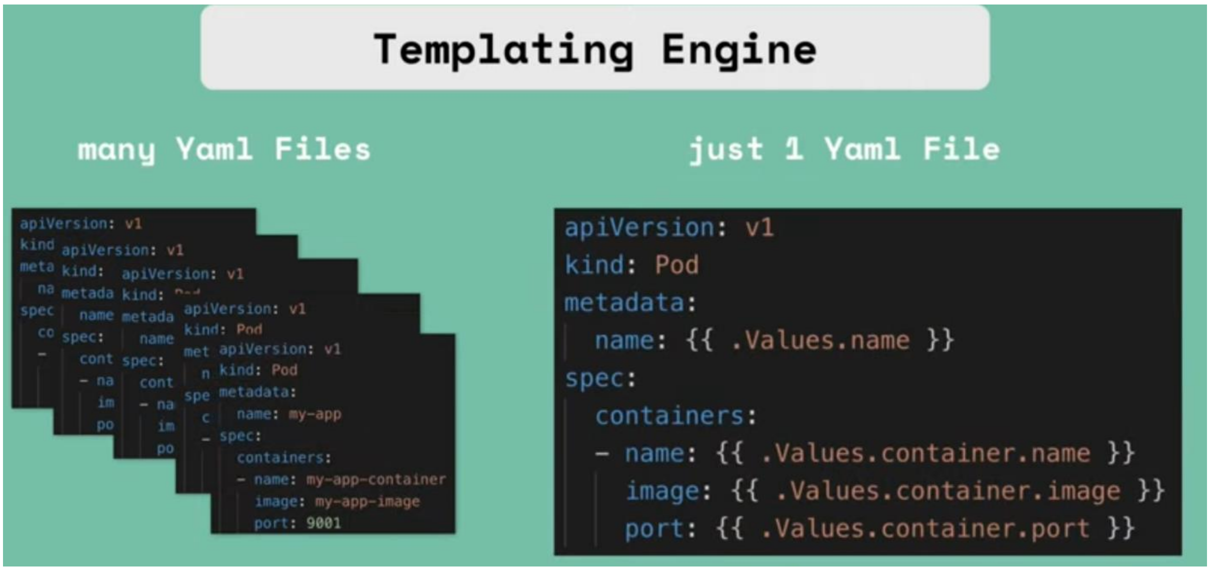


```
apiVersion: v1
kind: Pod
metadata:
  name: {{ .Values.name }}
spec:
  containers:
  - name: {{ .Values.container.name }}
    image: {{ .Values.container.image }}
    port: {{ .Values.container.port }}
```



```
name: my-app
container:
  name: my-app-container
  image: my-app-image
  port: 9001
```

```
apiVersion: v1
kind: Pod
metadata:
  name: {{ .Values.name }}
spec:
  containers:
  - name: {{ .Values.container.name }}
    image: {{ .Values.container.image }}
    port: {{ .Values.container.port }}
```



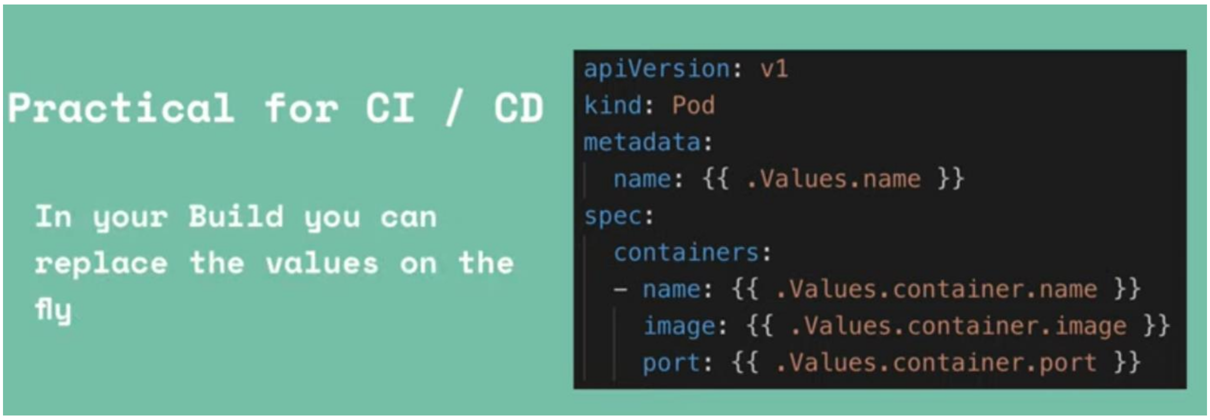
Templating Engine

many Yaml Files

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
  - name: my-app-container
    image: my-app-image
    port: 9001
```

just 1 Yaml File

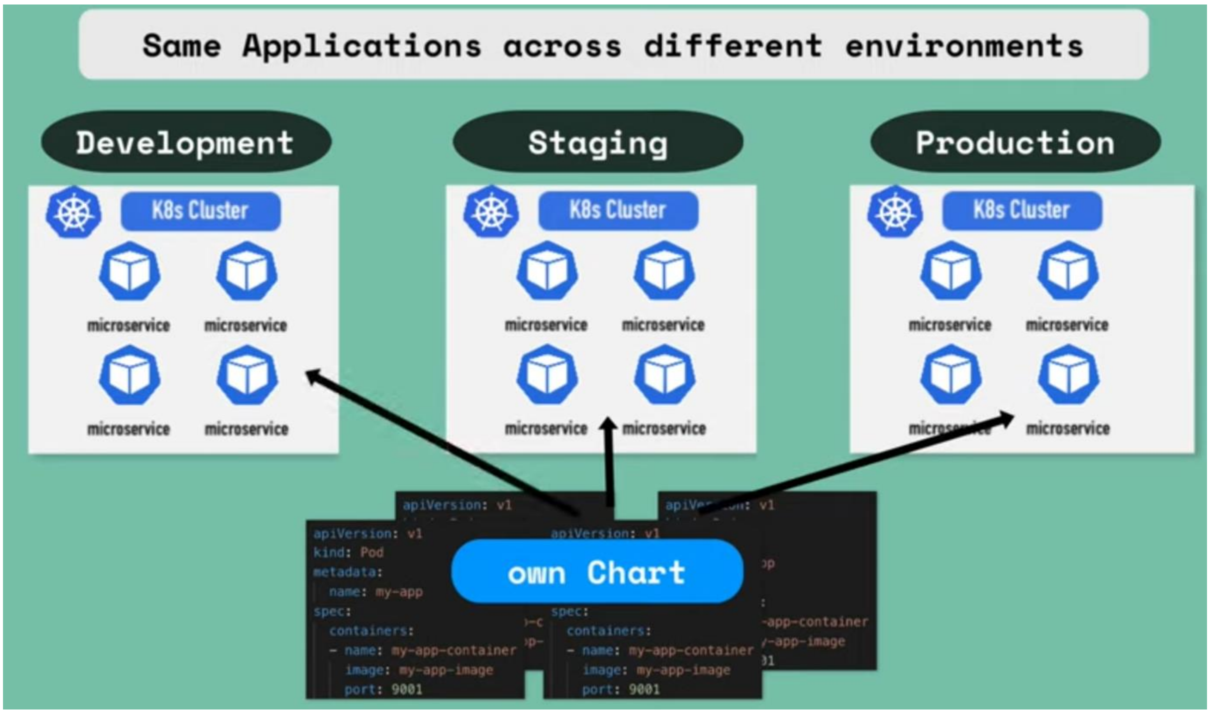
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: {{ .Values.name }}
spec:
  containers:
  - name: {{ .Values.container.name }}
    image: {{ .Values.container.image }}
    port: {{ .Values.container.port }}
```



Practical for CI / CD

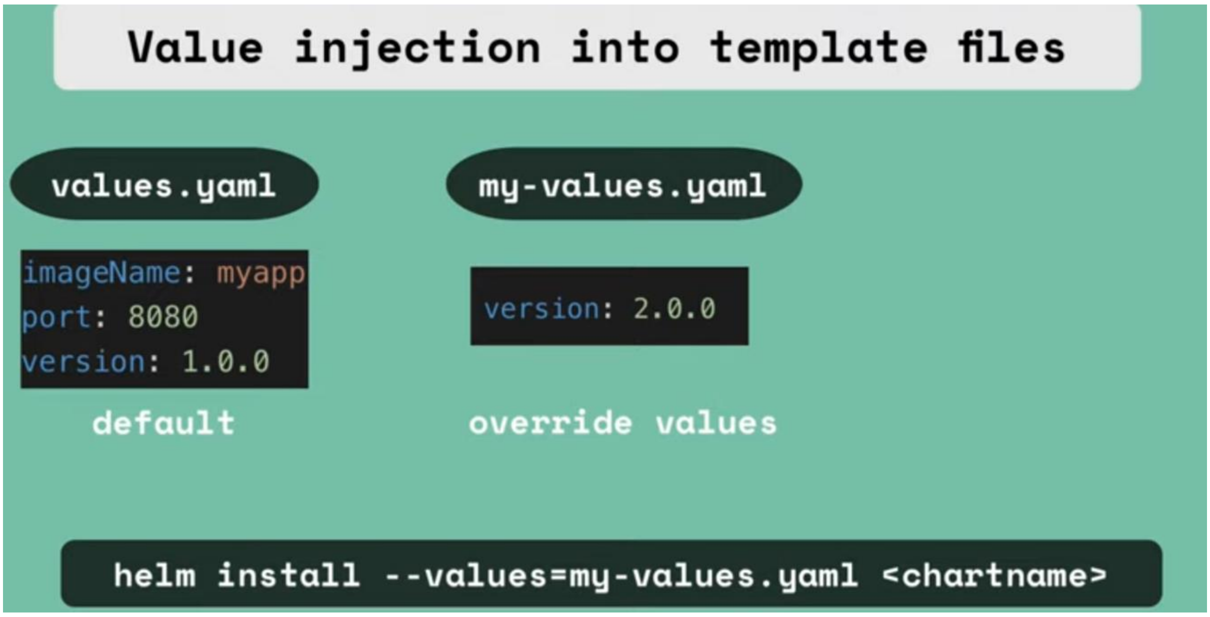
In your Build you can replace the values on the fly

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: {{ .Values.name }}
spec:
  containers:
  - name: {{ .Values.container.name }}
    image: {{ .Values.container.image }}
    port: {{ .Values.container.port }}
```



Same Applications across different environments

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
  - name: my-app-container
    image: my-app-image
    port: 9001
```



Value injection into template files

values.yaml

```
imageName: myapp
port: 8080
version: 1.0.0
```

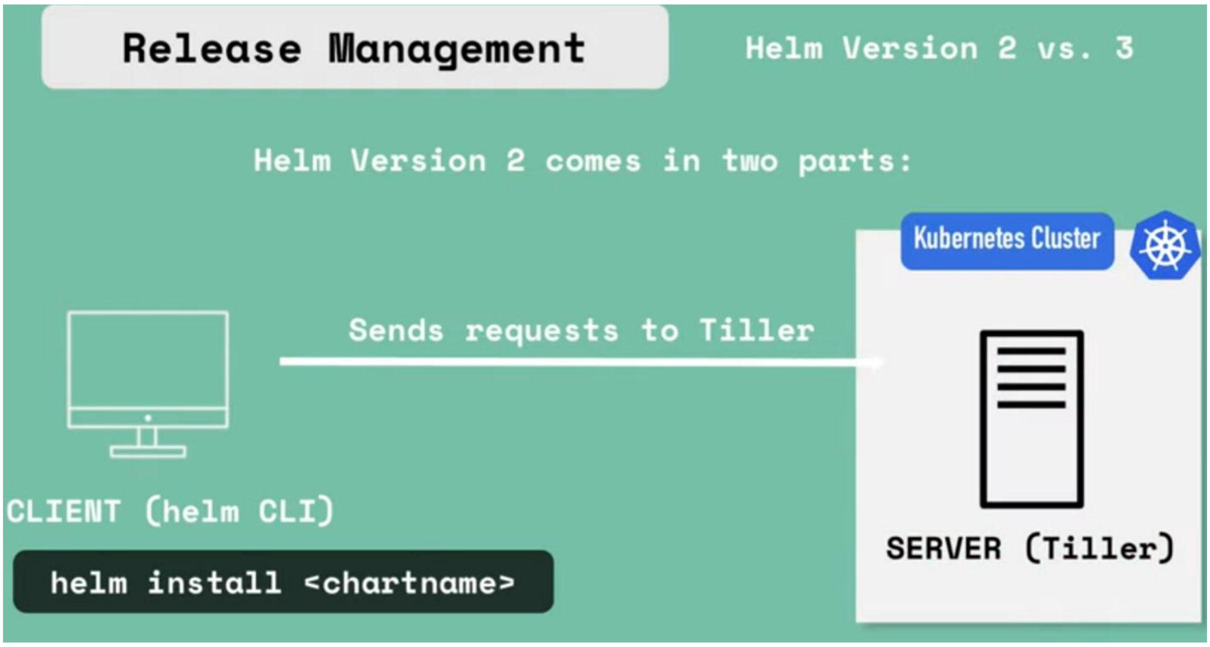
default

my-values.yaml

```
version: 2.0.0
```

override values

```
helm install --values=my-values.yaml <chartname>
```

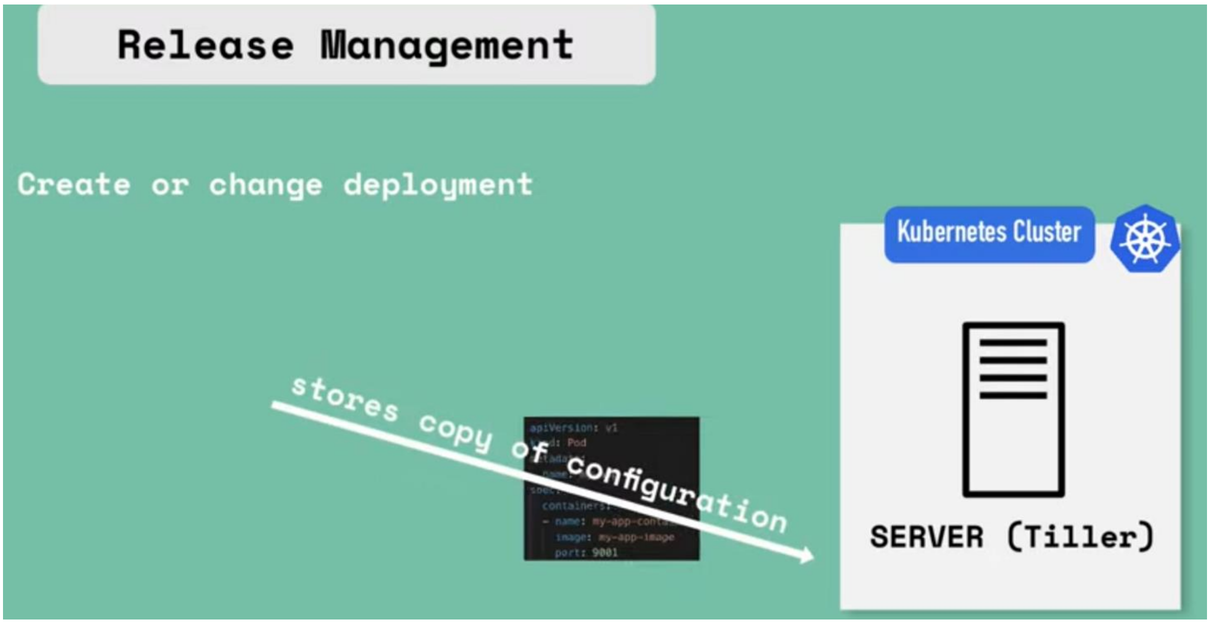


Release Management

Helm Version 2 vs. 3

Helm Version 2 comes in two parts:

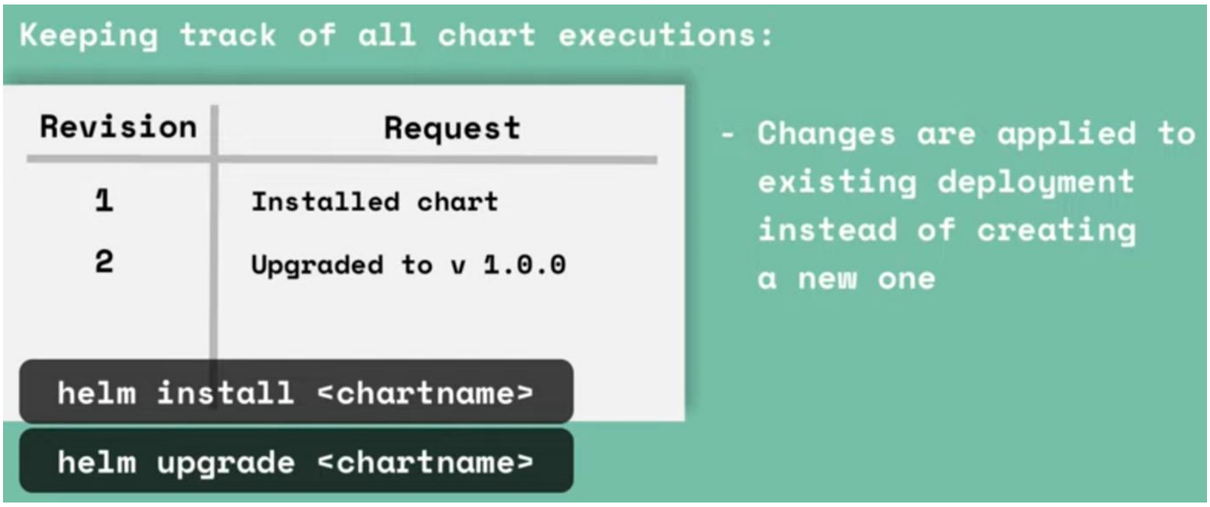
```
helm install <chartname>
```



Release Management

Create or change deployment

```
apiVersion: v1
kind: Pod
metadata:
  name:
spec:
  containers:
  - name: my-app-conta
    image: my-app-image
    port: 9001
```

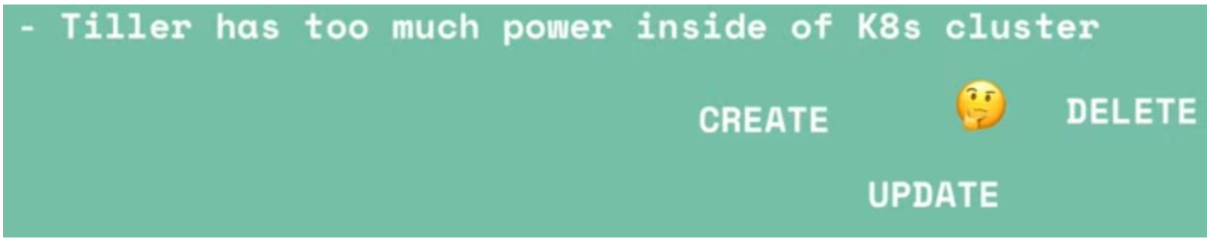


Keeping track of all chart executions:

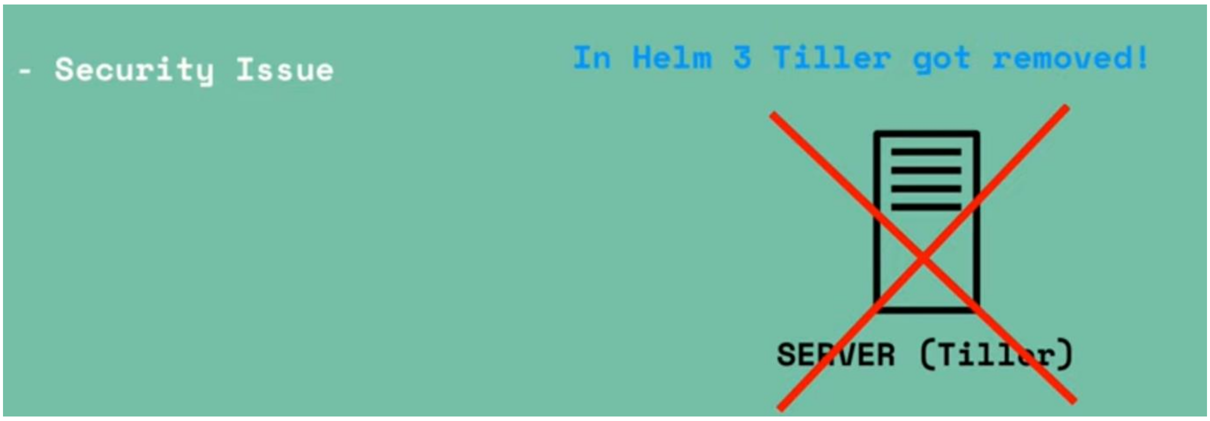
Revision	Request
1	Installed chart
2	Upgraded to v 1.0.0

```
helm install <chartname>
helm upgrade <chartname>
```

- Changes are applied to existing deployment instead of creating a new one

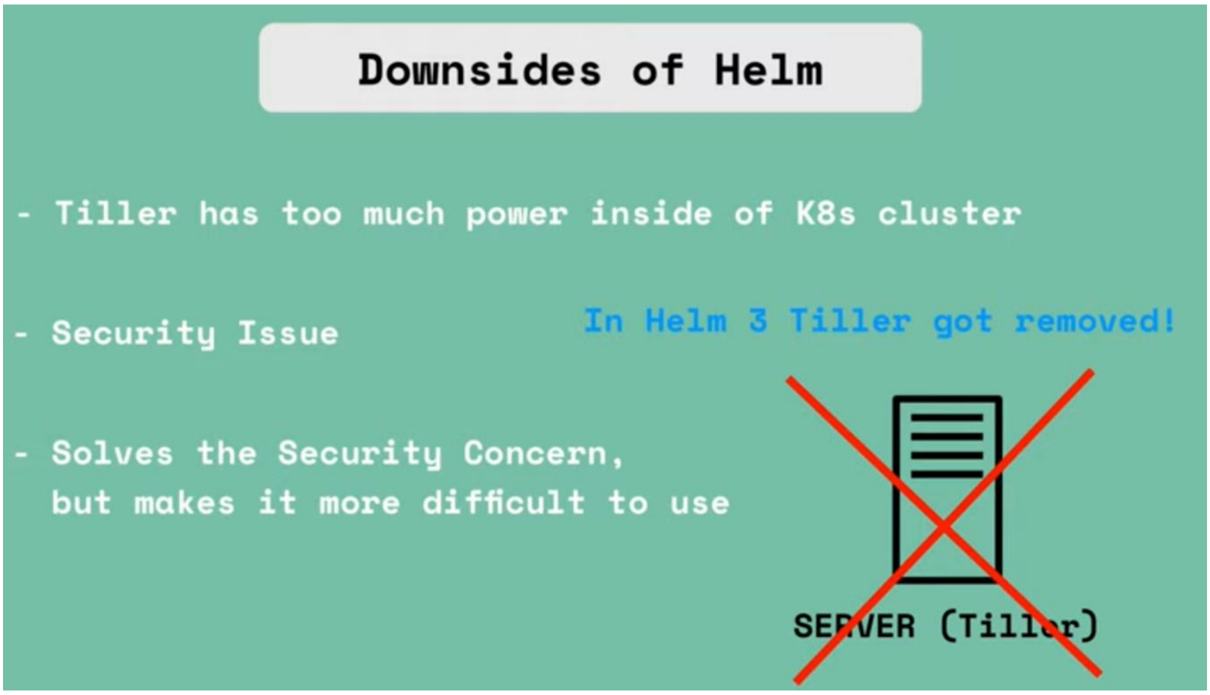


- Tiller has too much power inside of K8s cluster



- Security Issue

In Helm 3 Tiller got removed!

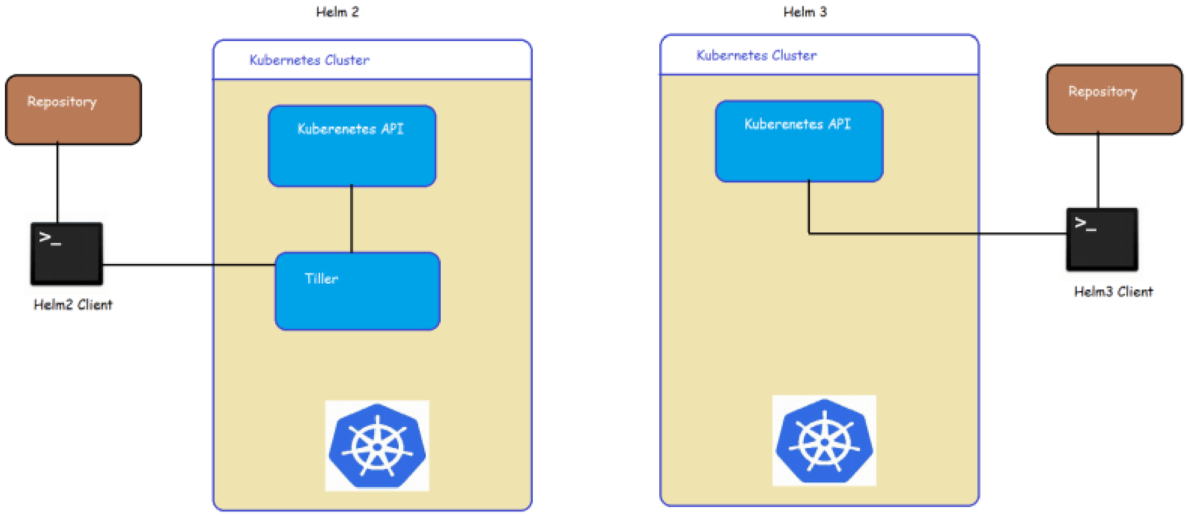


Downsides of Helm

- Tiller has too much power inside of K8s cluster
- Security Issue
- Solves the Security Concern, but makes it more difficult to use

In Helm 3 Tiller got removed!

Helm 2 vs Helm 3 Architecture



- Helm and k8s work like a client/server application.
- Helm client pushes the resources to the k8s cluster.
- In Helm 2 the server-side depends on the Tiller whereas Helm 3 got rid of Tiller and entirely relies on Kubernetes API
- Helm 3 authenticates and authorizes by taking the credentials kubectl

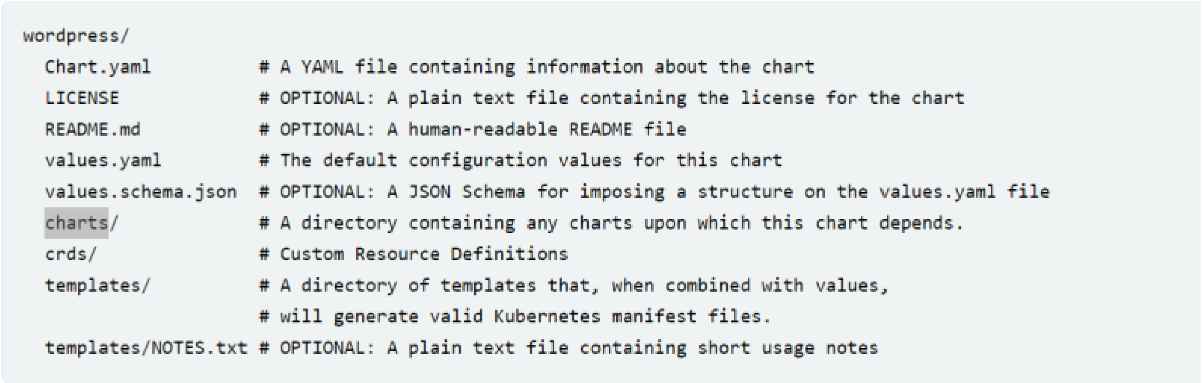
Quick YAML Refresher

- Multi line string

```
configurations: |
  server.port=8443
  logging.file.path=/var/log
```

Helm Chart Structure

- When we create a helm chart the directory structure will be as shown below



```
wordpress/
  Chart.yaml          # A YAML file containing information about the chart
  LICENSE             # OPTIONAL: A plain text file containing the license for the chart
  README.md           # OPTIONAL: A human-readable README file
  values.yaml         # The default configuration values for this chart
  values.schema.json  # OPTIONAL: A JSON Schema for imposing a structure on the values.yaml file
  charts/             # A directory containing any charts upon which this chart depends.
  crds/               # Custom Resource Definitions
  templates/          # A directory of templates that, when combined with values,
                      # will generate valid Kubernetes manifest files.
  templates/NOTES.txt # OPTIONAL: A plain text file containing short usage notes
```

- Now let’s try to understand the purpose of file/directory in the helm charts

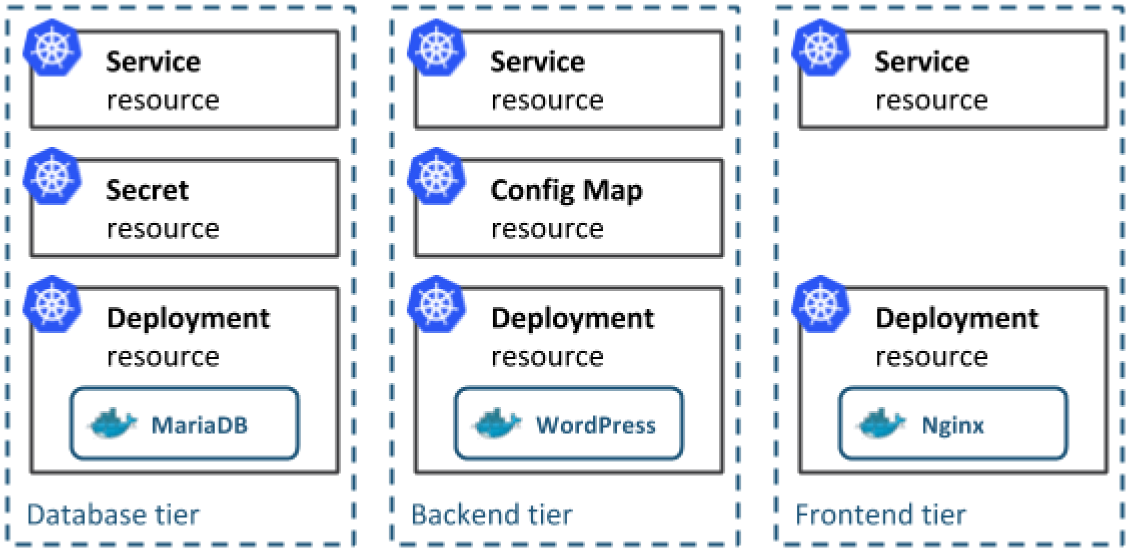
File/directory	Definition	Required
Chart.yaml	This file contains metadata about Helm chart	Yes
templates/	This directory contains K8s resources in YAML Format	Yes, unless dependencies are declared in Chart.yaml
template/NOTES.txt	A file that can be generated to provide usage instructions during chat installtions	No
values.yaml	This file contains the charts default values	No, but every chart should contain this file as a best practice
.helmignore	A file that consists of list of files and directories that should be ommited from Helm Charts packaging	No
charts/	This directory contains charts that the current Helm Chart depends on	Does not need to be explicitly provided as Helm's dependency management system will automatically create this directory
Chart.lock	A file used to save the previously applied dependency version	Does not need to be explicitly provided as Helm's dependency management system will automatically create this directory
crds/	A directory that contains Custom Resource Definition YAML resources to be installed before templates	No
README.md	A file that contains installation & usage information about Helm Chart	No, but every chart should contain this file as a best practice
LICENSE	A file that contains HELM charts LICENSE	No
values.schema.json	A file that contains the chart's value schema in JSON format	No

- Lets create a dummy chart

```
helm create --help
helm create helloworld
```

Creating Helm chart:

For a typical cloud-native application with a 3-tier architecture, the diagram below illustrates how it might be described in terms of Kubernetes objects. In this example, each tier consists of a Deployment and Service object, and may additionally define ConfigMap or Secret objects. Each of these objects are typically defined in separate YAML files, and are fed into the *kubectl* command line tool.



A Helm chart encapsulates each of these YAML definitions, provides a mechanism for configuration at deploy-time and allows you to define metadata and documentation that might be useful when sharing the package. Helm can be useful in different scenarios:

- Find and use popular software packaged as Kubernetes charts
- Share your own applications as Kubernetes charts
- Create reproducible builds of your Kubernetes applications
- Intelligently manage your Kubernetes object definitions
- Manage releases of Helm packages

Let's explore the second and third scenarios by creating our first chart.

Step 1: Generate your first chart

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

```
helm create mychart
```

Helm will create a new directory in your project called *mychart* with the structure shown below. Let's navigate our new chart (pun intended) to find out how it works.

```
mychart
|-- Chart.yaml
|-- charts
|-- templates
|   |-- NOTES.txt
|   |-- _helpers.tpl
|   |-- deployment.yaml
|   |-- ingress.yaml
|   `-- service.yaml
`-- values.yaml
```

Templates

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

It's worth noting however, that the directory is named *templates*, and Helm runs each file in this directory through a Go template rendering engine. Helm extends the template language, adding a number of utility functions for writing charts. Open the *service.yaml* file to see what this looks like:

```
apiVersion: v1
kind: Service
metadata:
name: {{ template "fullname" . }}
labels:
  chart: "{{ .Chart.Name }}-{{ .Chart.Version | replace "+" "_" }}"
spec:
type: {{ .Values.service.type }}
ports:
- port: {{ .Values.service.externalPort }}
  targetPort: {{ .Values.service.internalPort }}
  protocol: TCP
  name: {{ .Values.service.name }}
selector:
  app: {{ template "fullname" . }}
```

This is a basic Service definition using templating. When deploying the chart, Helm will generate a definition that will look a lot more like a valid Service. We can do a dry-run of a *helm install* and enable debug to inspect the generated definitions:

```
helm install --dry-run --debug ./mychart
...
# Source: mychart/templates/service.yaml
apiVersion: v1
kind: Service
metadata:
name: pouring-puma-mychart
labels:
  chart: "mychart-0.1.0"
spec:
type: ClusterIP
ports:
- port: 80
  targetPort: 80
  protocol: TCP
  name: nginx
selector:
  app: pouring-puma-mychart
...
```

Values

The template in *service.yaml* makes use of the Helm-specific objects *.Chart* and *.Values.*. The former provides metadata about the chart to your definitions such as the name, or version. The latter *.Values* object is a key element of Helm charts, used to expose configuration that can be set at the time of deployment. The defaults for this object are defined in the *values.yaml* file. Try changing the default value for *service.internalPort* and execute another dry-run, you should find that the *targetPort* in the Service and the *containerPort* in the Deployment changes. The *service.internalPort* value is used here to ensure that the Service and Deployment objects work together correctly. The use of templating can greatly reduce boilerplate and simplify your definitions.

If a user of your chart wanted to change the default configuration, they could provide overrides directly on the command-line:

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

For more advanced configuration, a user can specify a YAML file containing overrides with the *--values* option.

Helpers and other functions

The *service.yaml* template also makes use of partials defined in *_helpers.tpl*, as well as functions like *replace*. The Helm documentation has a deeper walkthrough of the templating language, explaining how functions, partials and flow control can be used when developing your chart.

Documentation

Another useful file in the *templates/* directory is the *NOTES.txt* file. This is a templated, plaintext file that gets printed out after the chart is successfully deployed. As we'll see when we deploy our first chart, this is a useful place to briefly describe the next steps for using a chart. Since *NOTES.txt* is run through the template engine, you can use templating to print out working commands for obtaining an IP address, or getting a password from a Secret object.

Metadata

As mentioned earlier, a Helm chart consists of metadata that is used to help describe what the application is, define constraints on the minimum required Kubernetes and/or Helm version and manage the version of your chart. All of this metadata lives in the *Chart.yaml* file. The Helm documentation describes the different fields for this file.

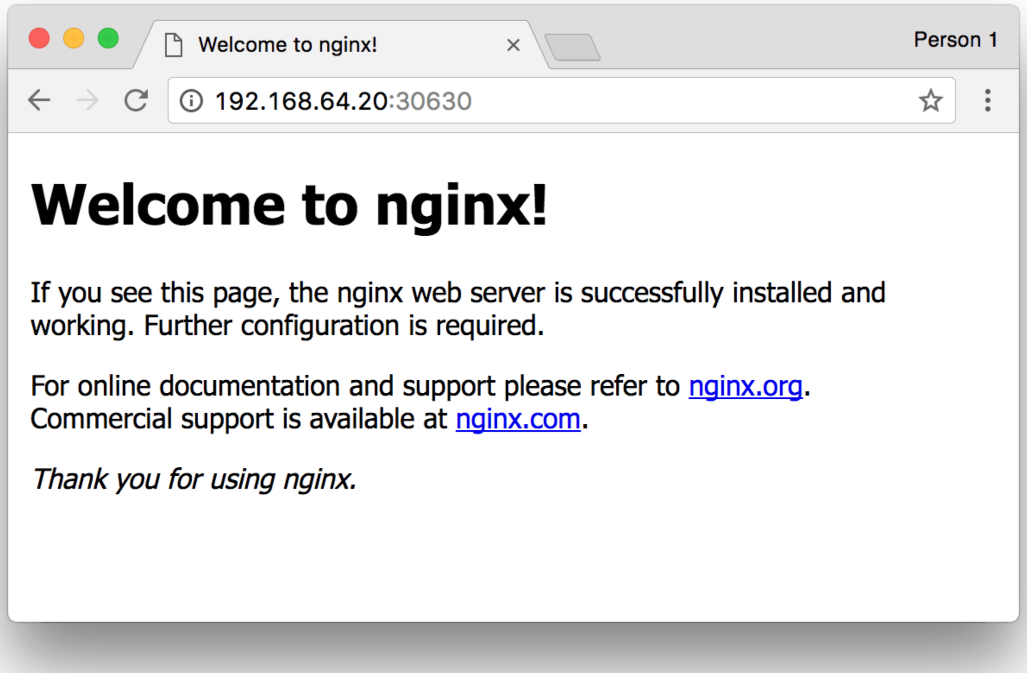
Step 2: Deploy your first chart

The chart you generated in the previous step is set up to run an NGINX server exposed via a Kubernetes Service. By default, the chart will create a *ClusterIP* type Service, so NGINX will only be exposed internally in the cluster. To access it externally, we'll use the *NodePort* type instead. We can also set the name of the Helm release so we can easily refer back to it. Let's go ahead and deploy our NGINX chart using the *helm install* command:

```
helm install example ./mychart --set service.type=NodePort
NAME:   example
LAST DEPLOYED: Tue May  2 20:03:27 2017
```

```
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Service
NAME           CLUSTER-IP  EXTERNAL-IP  PORT(S)       AGE
example-mychart  10.0.0.24   <nodes>      80:30630/TCP  0s

==> v1beta1/Deployment
NAME           DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
example-mychart  1        1        1           0          0s


NOTES:
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 output of *helm install* displays a handy summary of the state of the release, what objects were created, and the rendered *NOTES.txt* file to explain what to do next. Run the commands in the output to get a URL to access the NGINX service and pull it up in your browser.



Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

If all went well, you should see the NGINX welcome page as shown above. Congratulations! You've just deployed your very first service packaged as a Helm chart!

Step 3: Modify chart to deploy a custom service

The generated chart creates a Deployment object designed to run an image provided by the default values. This means all we need to do to run a different service is to change the referenced image in *values.yaml*.

We are going to update the chart to run a todo list application available on Docker Hub. In *values.yaml*, update the image keys to reference the todo list image:

```
image:
repository: prydonius/todo
tag: 1.0.0
pullPolicy: IfNotPresent
```

As you develop your chart, it's a good idea to run it through the linter to ensure you're following best practices and that your templates are well-formed. 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
```

The linter didn't complain about any major issues with the chart, so we're good to go. However, as an example, here is what the linter might output if you managed to get something wrong:

```
echo "malformed" > mychart/values.yaml
helm lint ./mychart
==> Linting mychart
[INFO] Chart.yaml: icon is recommended
[ERROR] values.yaml: unable to parse YAML
   error converting YAML to JSON: yaml: line 34: could not find expected ':'

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

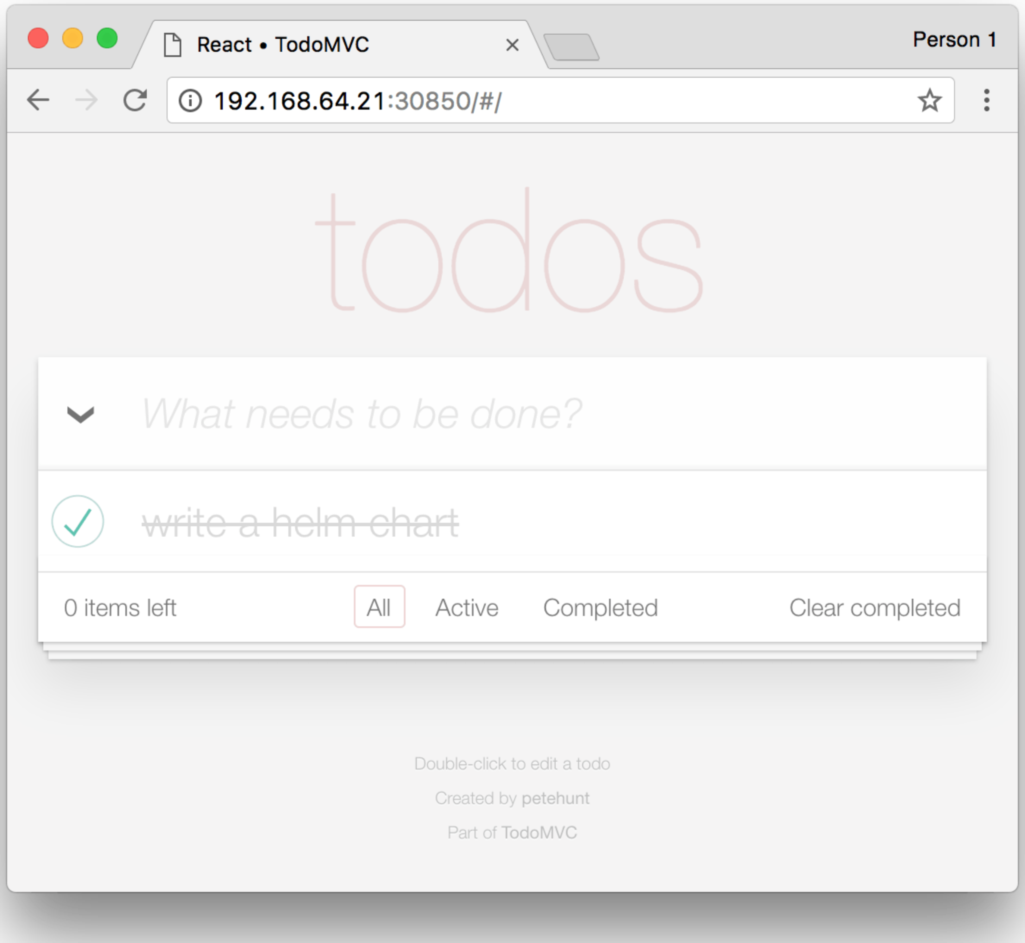
This time, the linter tells us that it was unable to parse my *values.yaml* file correctly. With the line number hint, we can easily find the fix the bug we introduced.

Now that the chart is once again valid, run *helm install* again to deploy the todo list application:

```
helm install example2 ./mychart --set service.type=NodePort
NAME:   example2
LAST DEPLOYED: Wed May  3 12:10:03 2017
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Service
NAME             CLUSTER-IP  EXTERNAL-IP  PORT(S)       AGE
example2-mychart  10.0.0.78   <nodes>      80:31381/TCP  0s
```

```
==> apps/v1/Deployment
NAME            DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
example2-mychart  1        1        1           0          0s


NOTES:
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 the commands in the NOTES to get a URL to access our application.



192.168.64.21:30850/#/

What needs to be done?

~~write a helm chart~~

0 items left

Double-click to edit a todo

Created by petehunt

Part of TodoMVC

If you have already built containers for your applications, you can run them with your chart by updating the default values or the *Deployment* template. Check out the Bitnami Docs for an introduction to containerizing your applications.

Step 4: Package it all up to share

So far in this tutorial, we've been using the *helm install* command to install a local, unpacked chart. However, if you are looking to share your charts with your team or the community, your consumers will typically install the charts from a tar package. We can use *helm package* to create the tar package:

```
helm package ./mychart
```

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

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

Repositories

In order to make it much easier to share packages, Helm has built-in support for installing packages from an HTTP server. Helm reads a repository index hosted on the server which describes what chart packages are available and where they are located.

We can use the *helm serve* command to run a local repository to serve our chart.

```
helm serve
Regenerating index. This may take a moment.
Now serving you on 127.0.0.1:8879
```

Now, in a separate terminal window, you should be able to see your chart in the local repository and install it from there:

```
helm search local
NAME          VERSION DESCRIPTION
local/mychart   0.1.0   A Helm chart for Kubernetes
```

```
helm install example4 local/mychart --set service.type=NodePort
```

To set up a remote repository you can follow the guide in the Helm documentation.

Dependencies

As the applications your packaging as charts increase in complexity, you might find you need to pull in a dependency such as a database. Helm allows you to specify sub-charts that will be created as part of the same release. To define a dependency, create a *requirements.yaml* file in the chart root directory:

```
cat > ./mychart/requirements.yaml <<EOF
dependencies:
- name: mariadb
version: 0.6.0
repository: https://charts.helm.sh/stable
EOF
```

Much like a runtime language dependency file (such as Python's *requirements.txt*), the *requirements.yaml* file allows you to manage your chart's dependencies and their versions. When updating dependencies, a lockfile is generated so that subsequent fetching of dependencies use a known, working version. Run the following command to pull in the MariaDB dependency we defined:

```
helm dep update ./mychart
Hang tight while we grab the latest from your chart repositories...
...Unable to get an update from the "local" chart repository (http://127.0.0.1:8879/charts):
   Get http://127.0.0.1:8879/charts/index.yaml: dial tcp 127.0.0.1:8879: getsockopt: connection
refused
...Successfully got an update from the "bitnami" chart repository
...Successfully got an update from the "incubator" chart repository
Update Complete. *Happy Helming!*
Saving 1 charts
Downloading mariadb from repo
$ ls ./mychart/charts
mariadb-0.6.0.tgz
```

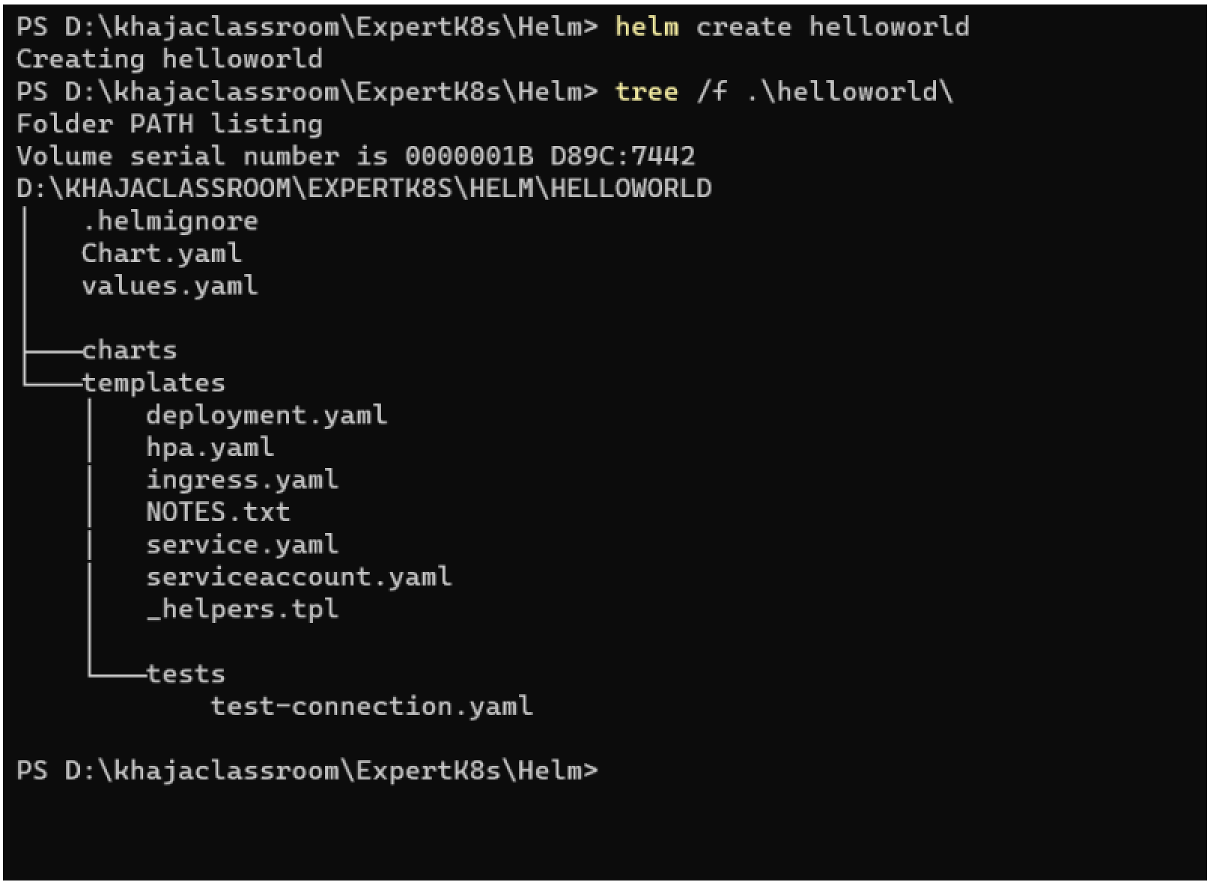
Helm has found a matching version in the *bitnami* repository and has fetched it into my chart's sub-chart directory. Now when we go and install the chart, we'll see that MariaDB's objects are created too:

```
helm install example5 ./mychart --set service.type=NodePort
NAME:   example5
LAST DEPLOYED: Wed May  3 16:28:18 2017
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Secret
NAME              TYPE    DATA  AGE
example5-mariadb  Opaque  2     1s

==> v1/ConfigMap
NAME              DATA  AGE
example5-mariadb  1     1s

==> v1/PersistentVolumeClaim
NAME              STATUS  VOLUME                                    CAPACITY  ACCESSMODES  AGE
example5-mariadb  Bound   pvc-229f9ed6-3015-11e7-945a-66fc987ccf32  8Gi       RWO          1s

==> v1/Service
NAME              CLUSTER-IP  EXTERNAL-IP  PORT(S)       AGE
example5-mychart  10.0.0.144  <nodes>      80:30896/TCP  1s
example5-mariadb  10.0.0.108  <none>       3306/TCP      1s

==> apps/v1/Deployment
NAME              DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
example5-mariadb  1        1        1           0          1s
example5-mychart  1        1        1           0          1s
```

NOTES:
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/

Contribute to the Bitnami repository!

As a chart author, you can help to build out Bitnami's chart repository by improving existing charts or submitting new ones. Checkout https://kubeapps.com to see what's currently available and head to https://github.com/bitnami/charts to get involved.



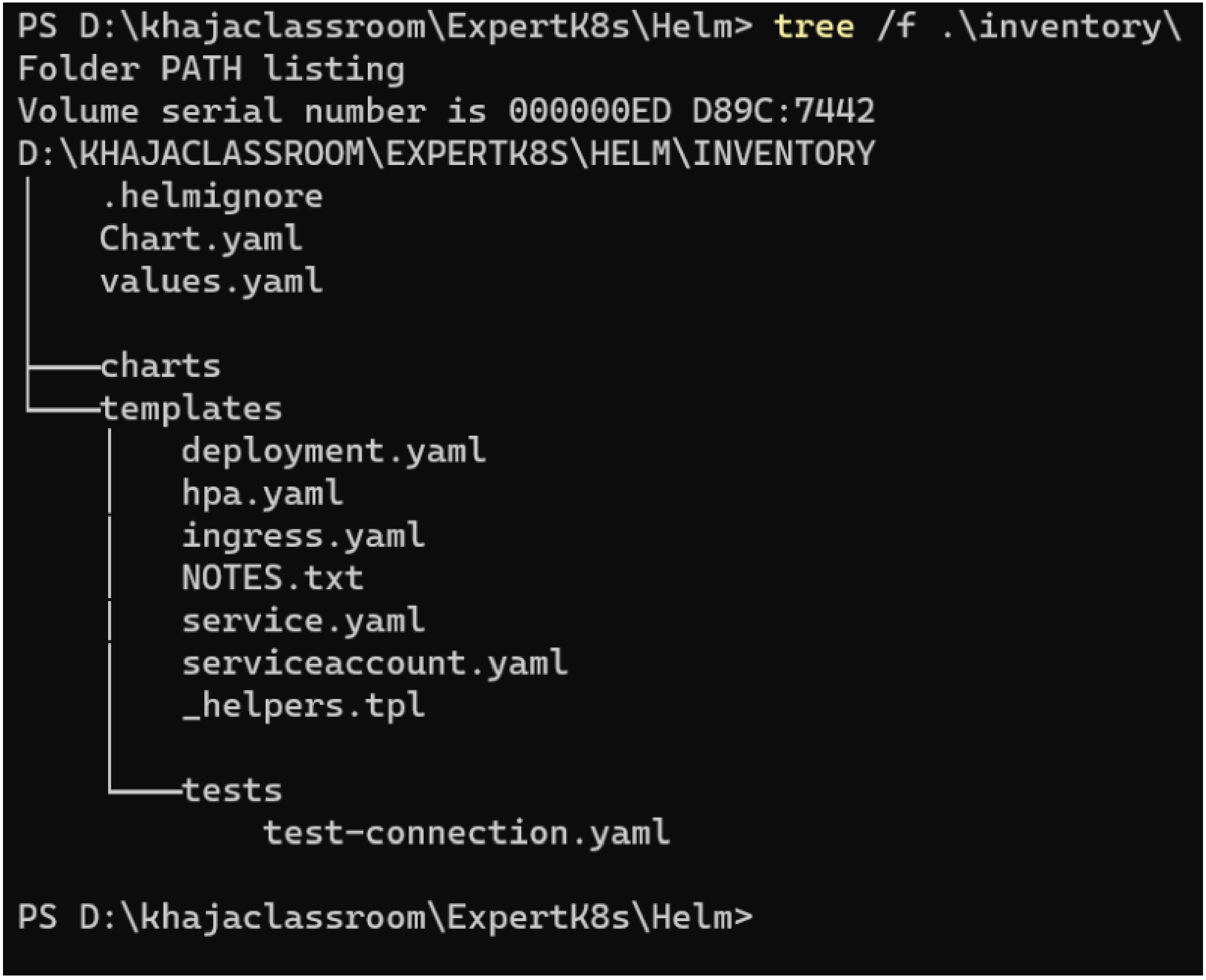
```
PS D:\khajaclassroom\ExpertK8s\Helm> helm create helloworld
Creating helloworld
PS D:\khajaclassroom\ExpertK8s\Helm> tree /f .\helloworld\
Folder PATH listing
Volume serial number is 0000001B D89C:7442
D:\KHAJACLASSROOM\EXPERTK8S\HELM\HELLOWORLD
│   .helmignore
│   Chart.yaml
│   values.yaml
│
├───charts
└───templates
    │   deployment.yaml
    │   hpa.yaml
    │   ingress.yaml
    │   NOTES.txt
    │   service.yaml
    │   serviceaccount.yaml
    │   _helpers.tpl
    │
    └───tests
            test-connection.yaml

PS D:\khajaclassroom\ExpertK8s\Helm>
```

Helm 2 VS helm 3:

Helm 2 has additional component call tiller which provide release management feature, but because of its ability to modify/create and update cluster, we need to remove it, as it was causing security concern.

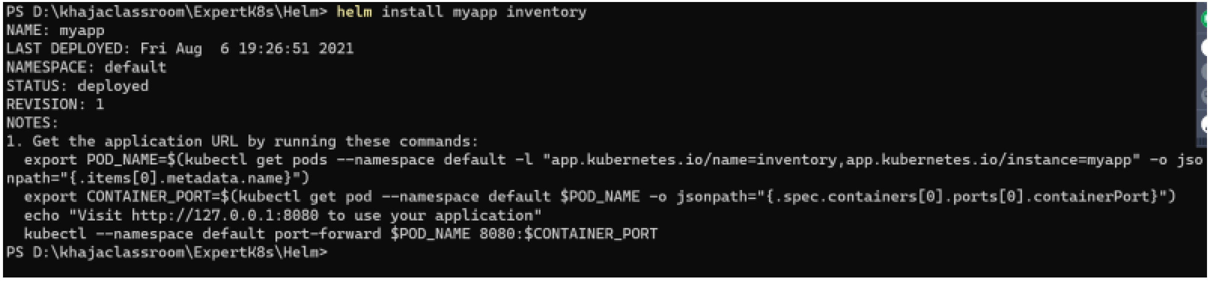
Helm Chart Creation Command

- Helm includes create command to make it easy for us to create charts
- This command creates a new **Nginx chart** with name of our choice



```
PS D:\khajaclassroom\ExpertK8s\Helm> tree /f .\inventory\
Folder PATH listing
Volume serial number is 000000ED D89C:7442
D:\KHAJACLASSROOM\EXPERTK8S\HELM\INVENTORY
│   .helmignore
│   Chart.yaml
│   values.yaml
│
├───charts
└───templates
    │   deployment.yaml
    │   hpa.yaml
    │   ingress.yaml
    │   NOTES.txt
    │   service.yaml
    │   serviceaccount.yaml
    │   _helpers.tpl
    │
    └───tests
            test-connection.yaml

PS D:\khajaclassroom\ExpertK8s\Helm>
```

- Lets try to install the chart which we have create



```
PS D:\khajaclassroom\ExpertK8s\Helm> helm install myapp inventory
NAME: myapp
LAST DEPLOYED: Fri Aug  6 19:26:51 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
1. Get the application URL by running these commands:
  export POD_NAME=$(kubectl get pods --namespace default -l "app.kubernetes.io/name=inventory,app.kubernetes.io/instance=myapp" -o jso
npath="{.items[0].metadata.name}")
  export CONTAINER_PORT=$(kubectl get pod --namespace default $POD_NAME -o jsonpath="{.spec.containers[0].ports[0].containerPort}")
  echo "Visit http://127.0.0.1:8080 to use your application"
  kubectl --namespace default port-forward $POD_NAME 8080:$CONTAINER_PORT
PS D:\khajaclassroom\ExpertK8s\Helm>
```

- The output from the command is

NAME: myapp
LAST DEPLOYED: Fri Aug 6 19:26:51 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1

helm install [NAME] [CHART]

```
NOTES:
1. Get the application URL by running these commands:
  export POD_NAME=$(kubectl get pods --namespace default -l
"app.kubernetes.io/name=inventory,app.kubernetes.io/instance=myapp" -o
jsonpath="{.items[0].metadata.name}")
  export CONTAINER_PORT=$(kubectl get pod --namespace default $POD_NAME -o
jsonpath="{.spec.containers[0].ports[0].containerPort}")
  echo "Visit http://127.0.0.1:8080 to use your application"
  kubectl --namespace default port-forward $POD_NAME 8080:$CONTAINER_PORT
```

- Now refer the charts.yaml https://helm.sh/docs/topics/charts/ for the offical docs
- In this charts.yaml lets try to focus on some name value pairs
  - apiVersion: v2: This tells helm what structure of chart we are using. An apiVersion of v2 is designed for Helm3
  - name: inventory: The name used to identify the chart
  - version: 0.1.0: Charts can have many versions. Helm uses the version information to order and identify charts
- Charts.yaml also contain descriptive information
  - home: URL of chart or projects
  - icon: an image in the form of URL
  - maintainers: contains list of maintainers
  - keywords: can hold list of keyworkds about the project
  - sources: list of URLs for the source code for project or chart
- Refer below for the sample chart.yaml

```
annotations:
  category: Database
apiVersion: v2
appVersion: 8.0.27
dependencies:
  - name: common
    repository: https://charts.bitnami.com/bitnami
    tags:
      - bitnami-common
    version: 1.x.x
description: Chart to create a Highly available MySQL cluster
engine: gotpl
home: https://github.com/bitnami/charts/tree/master/bitnami/mysql
icon: https://bitnami.com/assets/stacks/mysql/img/mysql-stack-220x234.png
keywords:
  - mysql
  - database
```

Link: https://github.com/bitnami/charts/blob/master/bitnami/mysql/Chart.yaml

Modifying Templates

- Helm is written in Go programming language and Go includes template packages. Helm leverages the text template package as the foundation for its templates Refer https://pkg.go.dev/text/template
- {{ and }} are the opening and closing brackets to enter and exit the template logic
- Sample

product: {{ .Values.product | default "kubernetes" | quote }}

- There are three parts to the template logic sepearted by a |. This is called as pipeline and works exactly in the sameway as pipeline in Unix/Linux Based system. The value or output of a function on the left is passed as a last argument to the next item in pipeline.
- .Values.product This comes for the data passed in when the templates are rendered
- This value is passed as last argument to the default function
- The default is the helm function and output of default is passed to the quoted
- The . at the start .Values.production is considered as root object in the current scope

Developing Templates

- Helm uses the Go text template engine provided as part of standard Go Libarary
- Actions:
  - Logic, control structures and data evaluations are wrapped by {{ and }}. These are called as actions.
  - Anything outside of actions is copied to output
  - When the curly braces are used to start and stop actions they can be accompanies by a – to remove leading or trailing white spaces.
- {{ "Hello" -}}, {{- "World" }}, {{- "of Helm" -}}
- # generated Output Hello,World,of Helm

Information Helm Passes to Templates

- When Helm renders a template it passes a single data object to the template with information you can access.
- Inside the template that object is representeed as . (i.e period)
- The properties on .values are specific to each chart based entirely on values in values.yaml
- What values should be present in values.yaml have no specific structure or schema.
- In addition to values, information about the release can be access as properties of .Release. This information includes
  - .Release.Name: name of the release
  - .Release.Namespace: Contains the namespace the chart is being released to
  - .Release.IsInstall: Set to true when relase is workload being installed
  - .Release.IsUpgrade: Set to true when the release is upgrade or rollback
  - .Release.Service: Lists the Service performing the release. when Helm installs a chart this value would be Helm
- The information in Chart.yaml can alos be found the data object at .Chart
  - .Chart.Name
  - .Chart.Version
  - .Chart.AppVersion
  - .Chart.Annotations

- Note the Names differ as names in Charts.yaml start with lowercase but Start with Uppercase later when they properties of .Chart object
- If you want to pass the custom information from the Chart.yaml to the template, you need to use annotations
- Helm also provides some data about the capabilities of the K8s cluster as properties of .Capabilities.
    - .Capabilities.ApiVersions: Contains the API Versions and resource types available in your cluster
    - .Capabilities.KubeVersion.Version: Full Kubernetes Version
    - .Capabilities.KubeVersion.Major: Contains major K8s version
    - .Capabilities.KubeVersion.Minor: The minor version of K8s being used in cluster
- The final piece of data passed into the template is details about the current template being executed.
    - .Template.Name: Contains the namespaced filepath to the template (inventory/templates/deployment.yaml)
    - .Template.BasePath: Contains the namespaced Path of Templates directory (inventory/templates)

Helm Pipelines

- A pipeline is a sequence of commands, functions and variables chained together

```
character: {{ .Values.character | default "Learning" | quote }}
```

Template Functions

- Within actions and pipelines, there are template functions which we can use.
- Functions provide a means to transform the data
- Most of the functions provided by helm are designed to be useful when generating charts.
- The functions range from simple like indent and nindent functions to indent output to complex ones that are able to reach into cluster and get information on current resources and resource types.
- Note: Most of the functions found in helm template are provided by a library named Sprig ref: http://masterminds.github.io/sprig/
- Helm templates have more than hundred function ref: https://helm.sh/docs/chart_template_guide/function_list/

Methods

- Helm also includes functions to detect the capabilities of K8s cluster and methods to work with files
- The .Capabilities object has the method .Capabilities.APIVersions.Has which takes a single argument for the K8s AP or type we want to check the existence of
- The other place where we can find methods is on .Files. It includes the following methods
  - .Files.Get name: Gets the content of the file name
  - .Files.GetBytes
  - .Files.Glob
  - .Files.AsConfig: Takes the file group and returns a flattened YAML suitable to include in the data section of K8s ConfigMap
  - .Files.AsSecret
  - .Files.Lines

Flow Control

- Go templates have if and else statements along with something similar but slightly different called with. if and else
- Lets assume in values.yaml file we have a section on ingress with enabled property

```
ingress:
  enabled: false
```

- In the ingress.yaml that creates the ingress resource for K8s the first and last lines are for the if statement

```
{{- if .Values.ingress.enabled  -}}
...
{{- end }}
```

Loops

- A sample list in yaml

```
characters:
  - ironman
  - thor
  - hulk

movies:
  avengers: 'This is good movie'
  wintersoldier: 'This is too good movie'
```

- We can generate above mentioned lists by using range

```
characters:
{{- range .Values.characters }}
  - {{ . | quote }}
{{- end }}
```

- We can generate a dictionary or map

```
movies:
{{- range $key, $value := .Values.movies }}
  - {{ $key }}: {{ $value | quote }}
```

Experiment- Creating a simple manifest for ngnix

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-example
spec:
  containers:
  - name: ubuntu
    image: ubuntu:trusty
    command: ["sleep"]
    args: ["1d"]

apiVersion: v1
kind: Pod
metadata:
  name: {{ .Values.pod.name }}
spec:
  containers:
  - name: ubuntu
    image: {{ .Values.image.repository | default "alpine" }}:{{ .Values.image.tag | default "latest" }}
    command: {{ .Values.container.command }}
    args: {{ .Values.container.args }}
```

The values

```yaml
# Default values for inventory.
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.

pod:
  name: "experiment-1"

image:
  repository: ubuntu
  pullPolicy: IfNotPresent
  # Overrides the image tag whose default is the chart appVersion.
  tag: "latest"

container:
  command: ["sleep"]
  args: ["1d"]
```

Exercise: Create a Helm Chart for k8s deployment- Write a Helm chart from scrath to create nginx-deployment (don’t use helm create) for k8s manifest

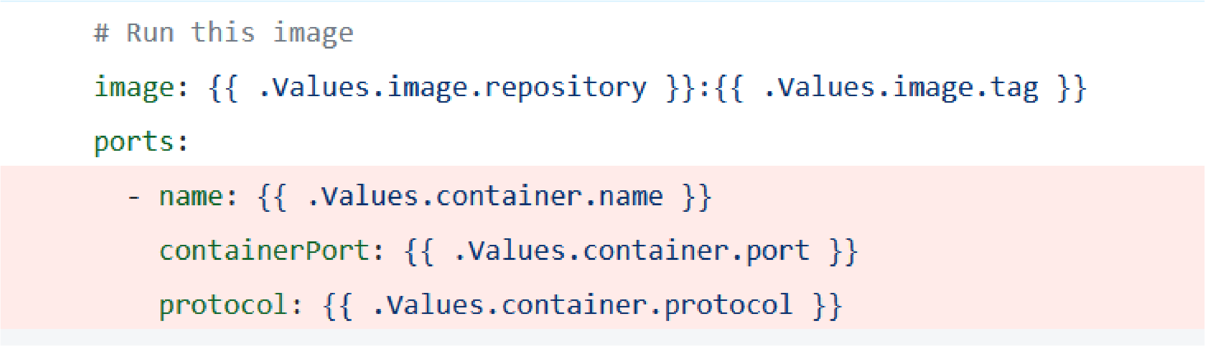
```
apiVersion: apps/v1
kind: Deployment
metadata:
  # Unique key of the Deployment instance
  name: deployment-example
spec:
  # 3 Pods should exist at all times.
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        # Apply this label to pods and default
        # the Deployment label selector to this value
        app: nginx
    spec:
      containers:
      - name: nginx
        # Run this image
        image: nginx:1.14
```

Get the values such as labels, container image, tag, ports replicas from values.yaml file. Ref: https://github.com/asquarezone/ExpertKubernetes/commit/e45194078e4f6ea1894c96ace66ad0a1c01f19bb for the solution

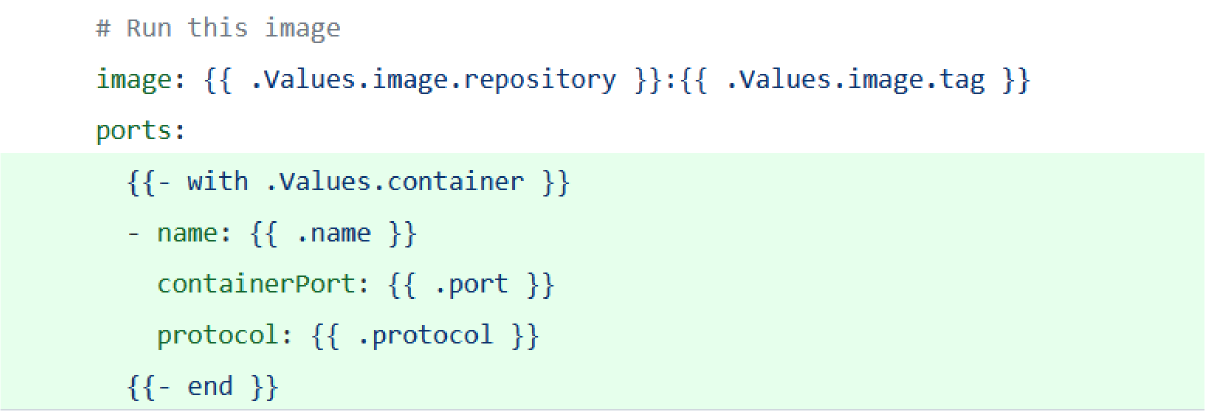
.

Helm with statement

- With in Helm in used to change the current scope.
- Now let’s look at created helm chart by using helm create command



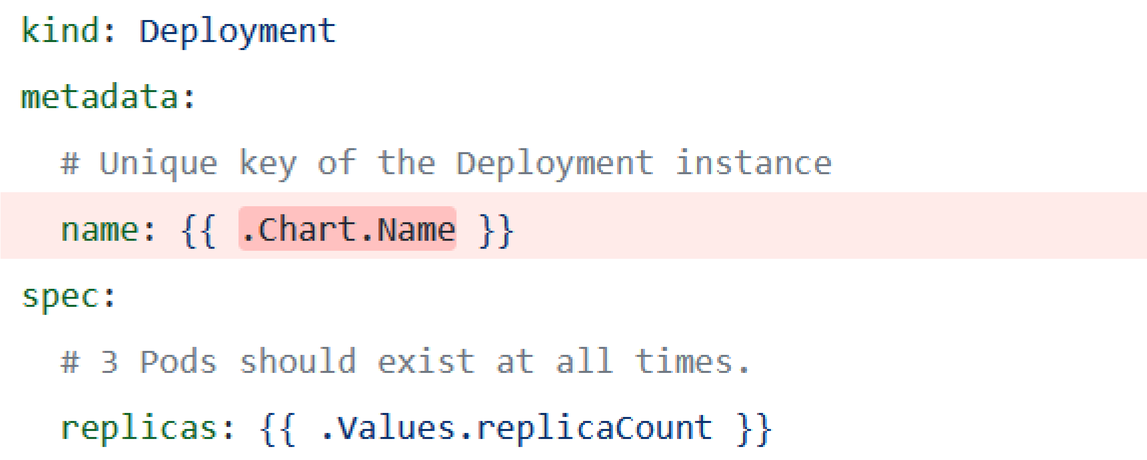
```
# Run this image
image: {{ .Values.image.repository }}:{{ .Values.image.tag }}
ports:
  - name: {{ .Values.container.name }}
    containerPort: {{ .Values.container.port }}
    protocol: {{ .Values.container.protocol }}
```



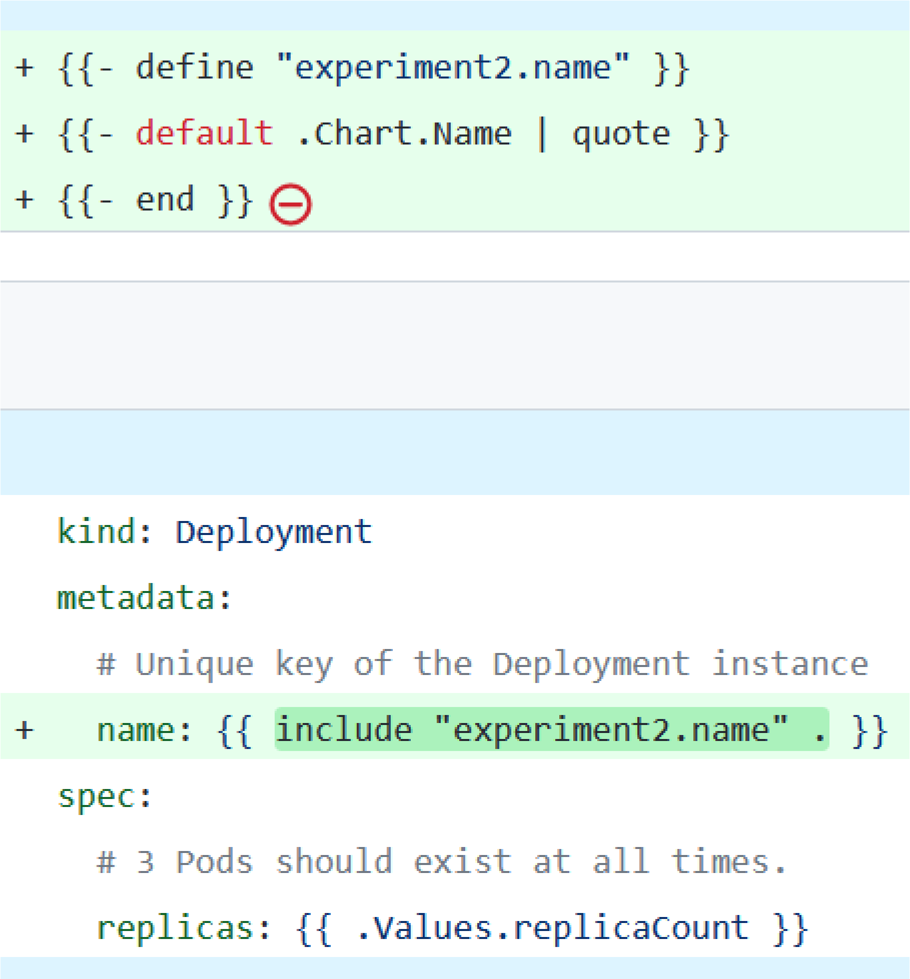
```
# Run this image
image: {{ .Values.image.repository }}:{{ .Values.image.tag }}
ports:
  {{- with .Values.container }}
  - name: {{ .name }}
    containerPort: {{ .port }}
    protocol: {{ .protocol }}
  {{- end }}
```

Named Templates

- Ref: https://helm.sh/docs/chart_template_guide/named_templates/ for the official docs



```
kind: Deployment
metadata:
  # Unique key of the Deployment instance
  name: {{ .Chart.Name }}
spec:
  # 3 Pods should exist at all times.
  replicas: {{ .Values.replicaCount }}
```



```
+ {{- define "experiment2.name" }}
+ {{- default .Chart.Name | quote }}
+ {{- end }} ⊖
```

```
  kind: Deployment
  metadata:
    # Unique key of the Deployment instance
+   name: {{ include "experiment2.name" . }}
  spec:
    # 3 Pods should exist at all times.
    replicas: {{ .Values.replicaCount }}
```