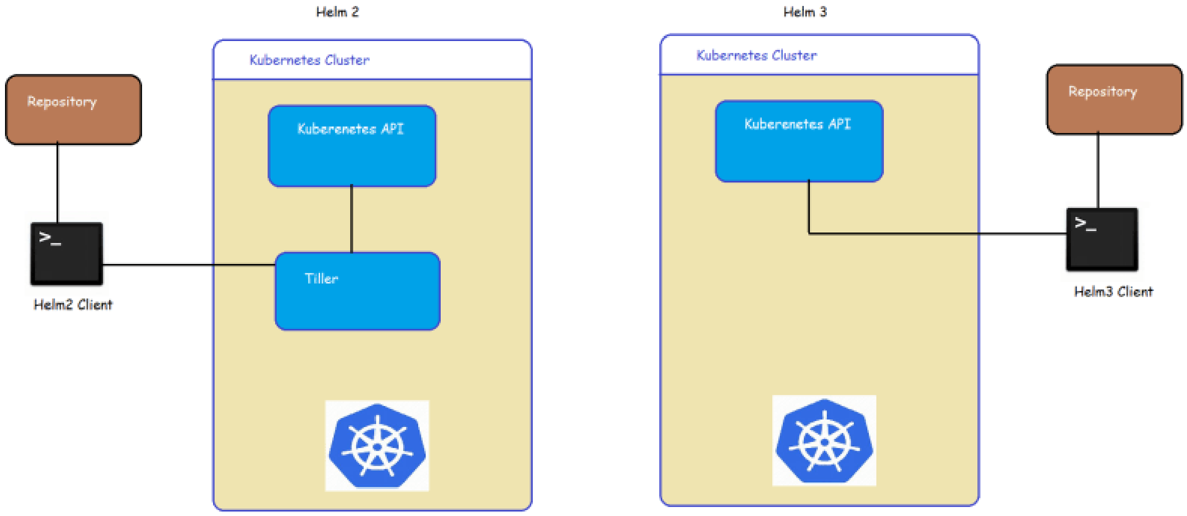
Helm 2 vs Helm 3 Architecture



- 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

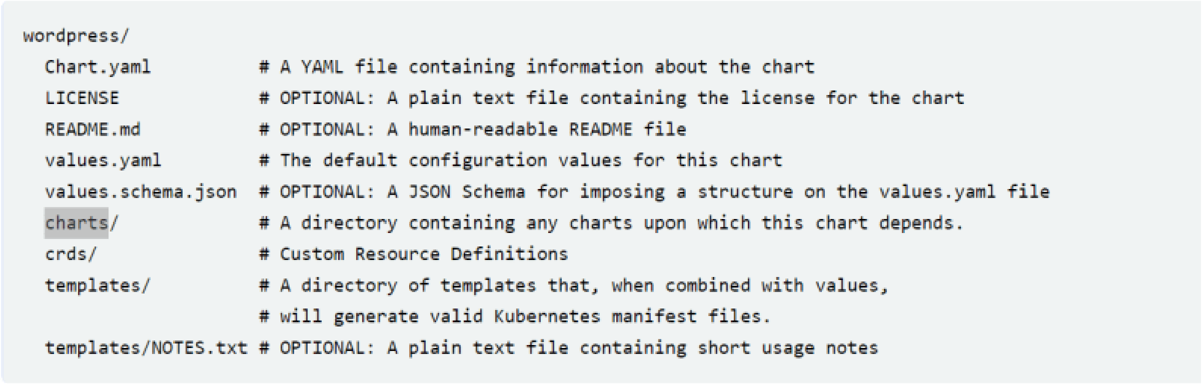
Quick YAML Refresher

- 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

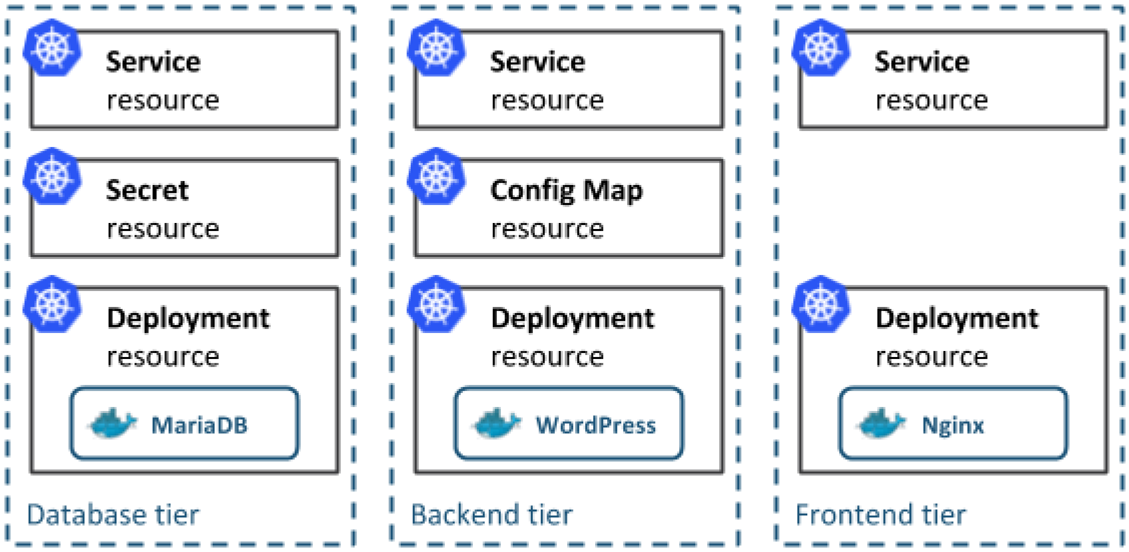
File/directory	Definition	Required
Chart.yaml	This file contains metadata about Helm chart	Yes
templates/	This directory contains K8s resources in YAML Format	Yes, unless dependencies are declared in Chart.yaml
template/NOTES.txt	A file that can be generated to provide usage instructions during chat 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.

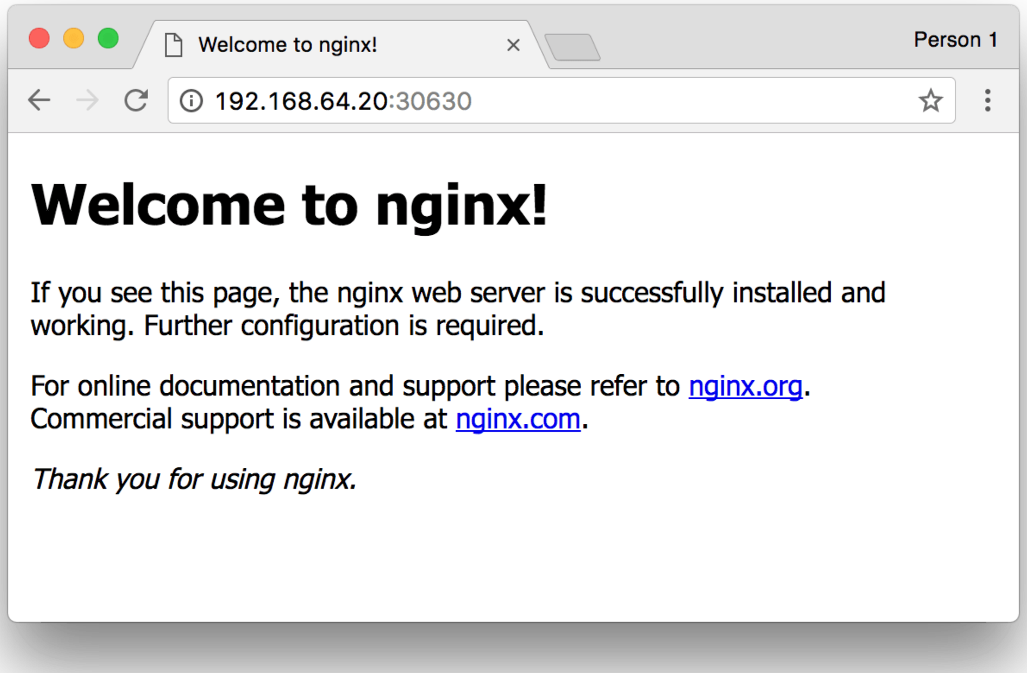
Step 2: Deploy your first chart

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
```

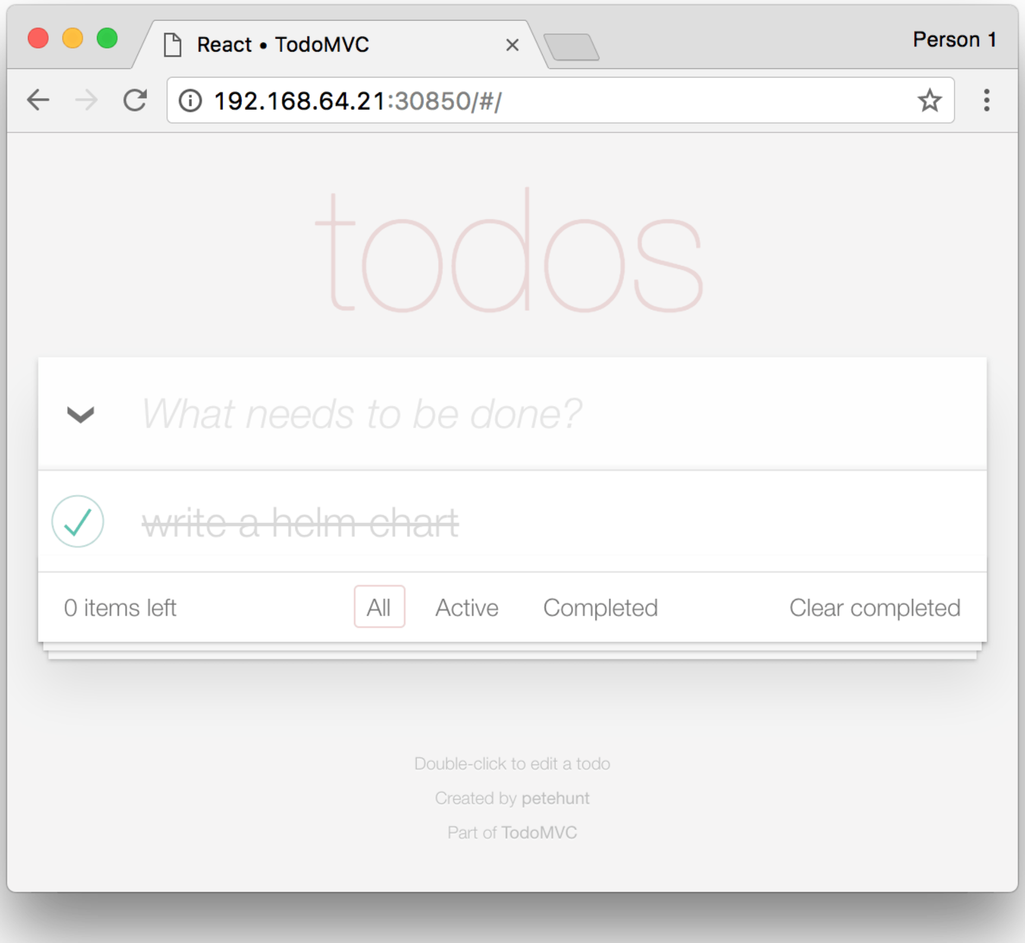
This time, the linter tells us that it was unable to parse my *values.yaml* file correctly. With the line number hint, we can easily find the fix the bug we introduced.

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.



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
```

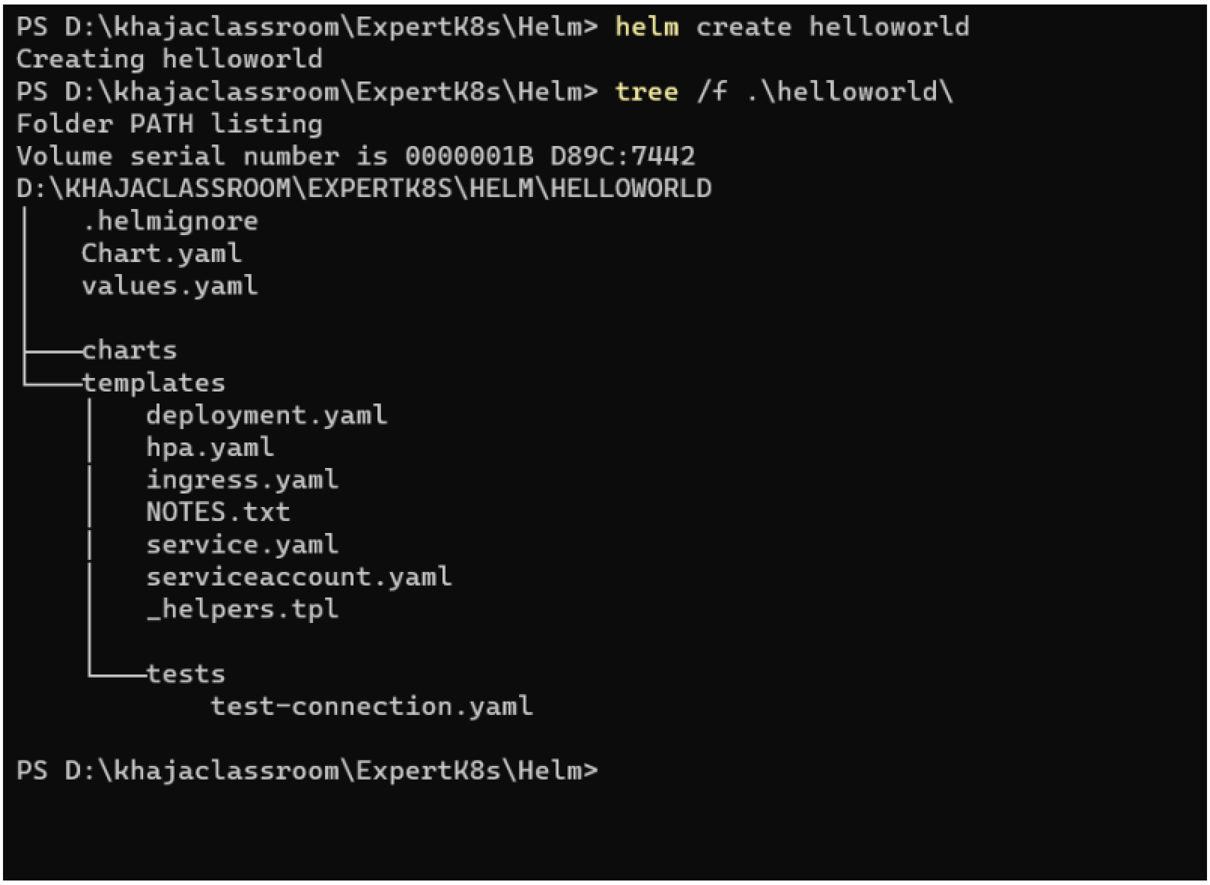
Helm has found a matching version in the *bitnami* repository and has fetched it into my chart's sub-chart directory. Now when we go and install the chart, we'll see that MariaDB's objects are created too:

```
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.