Using Helm and Kubernetes

**1. Overview**

[**Helm**](https://helm.sh/)**is a package manager for Kubernetes applications**. In this tutorial, we'll understand the basics of Helm and how they form a powerful tool for working with Kubernetes resources.

Over the past years, Kubernetes has grown tremendously, and so has the ecosystem supporting it. Recently, Helm has been announced as an incubating project by [Cloud Native Computing Foundation (CNCF)](https://www.cncf.io/), which shows its growing popularity amongst Kubernetes users.

**2. Background**

Although these terms are fairly common these days, particularly amongst those working with cloud technologies, let's go through them quickly for those unaware:

1. [Container](https://www.docker.com/resources/what-container): **Container refers to operating system level virtualization**. Multiple containers run within an operating system in isolated user-spaces. Programs running within a container have access only to resources assigned to the container.
2. [Docker](https://www.docker.com/): **Docker is a popular program to create and run containers**. It comes with Docker Daemon, which is the main program managing containers. Docker Daemon offers access to its features through Docker Engine API, which is used by Docker Command-Line Interface (CLI). [Please refer to this article for a more detailed description of Docker](https://www.baeldung.com/dockerizing-spring-boot-application).
3. [Kubernetes](https://kubernetes.io/): **Kubernetes is a popular container orchestration program**. Although it is designed to work with different containers, Docker is most often used. It offers a wide selection of features including deployment automation, scaling, and operations across a cluster of hosts. [There is excellent coverage of Kubernetes in this article for further reference](https://www.baeldung.com/kubernetes).

**3. Helm Architecture**

Helm has a fairly simple architecture, which is comprised of a client and an in-cluster server:

* Tiller Server: **Helm manages Kubernetes application through a component called Tiller Server** installed within a Kubernates cluster. Tiller interacts with the Kubernetes API server to install, upgrade, query and remove Kubernetes resources.
* Helm Client: **Helm provides a command-line interface for users to work with Helm Charts**. Helm Client is responsible for interacting with the Tiller Server to perform various operations like install, upgrade and rollback charts.

**4. Helm Charts**

**Helm manages Kubernetes resource packages through Charts**.

We'll see more about charts as we create them shortly, but for now, a chart is nothing but a set of information necessary to create a Kubernetes application, given a Kubernetes cluster:

* A **chart is a collection of files** organized in a specific directory structure
* The configuration information related to a chart is managed in the configuration
* Finally, **a running instance of a chart with a specific config is called a release**

**5. Set-up**

We'll need a few things to be set-up beforehand for us to develop our first Helm Chart.

Firstly, to begin working with Helm, we need a Kubernetes cluster. For this tutorial, we'll use **[Minikube](https://kubernetes.io/docs/setup/minikube/), which offers an excellent way to work with a single-node Kubernetes cluster locally**. On Windows, it's now possible to use Hyper-V as the native Hypervisor to run Minikube. [Refer to this article to understand setting up Minikube in more details](https://www.baeldung.com/spring-boot-minikube).

And, we'll need a basic application to manage within the Kubernetes cluster. For this tutorial, we'll use a simple Spring Boot application packaged as a Docker container. [For a more detailed description of how to package such an application as a Docker container, please refer to this article](https://www.baeldung.com/dockerizing-spring-boot-application#Dockerize).

**6. Installing Helm**

There are several ways to install Helm that are neatly described on [the official install page on Helm](https://helm.sh/docs/using_helm/#installing-helm). **The quickest way to install helm on Windows is using**[**Chocolaty**](https://chocolatey.org/), a package manager for Windows platforms.

Using Chocolaty, it is a simple one-line command to install Helm:

|  |  |
| --- | --- |
| 1 | choco install kubernetes-helm |

This installs the Helm Client locally.

Now, we need to initialize Helm CLI, which effectively also installs the Tiller Server onto a Kubernetes cluster as identified through the Kubernetes configuration. Please **ensure that the Kubernetes cluster is running and accessible through *kubectl* before initializing Helm**:

|  |  |
| --- | --- |
| 1 | kubectl cluster-info |

And then, we can initialize Helm through the Helm CLI itself:

|  |  |
| --- | --- |
| 1 | helm init |

**7. Developing Our First Chart**

Now we are ready to develop our first Helm Chart with templates and values.

**7.1. Creating a Chart**

**Helm CLI, which we installed earlier, is quite handy in creating a chart**:

|  |  |
| --- | --- |
| 1 | helm create hello-world |

Please note that the **name of the chart provided here will be the name of the directory where the chart is created** and stored.

Let's quickly see the directory structure created for us:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | hello-world /    Chart.yaml    values.yaml    templates /    charts /    .helmignore |

Let's understand the relevance of these files and folders created for us:

* *Chart.yaml*: This is the main file that contains the description of our chart
* *values.yaml*: this is the file that contains the default values for our chart
* *templates*: This is the directory where Kubernetes resources are defined as templates
* *charts*: This is an optional directory that may contain sub-charts
* *.helmignore*: This is where we can define patterns to ignore when packaging (similar in concept to .gitignore)

**7.2. Creating Template**

If we see inside the template directory, we'll notice that **few templates for common Kubernetes resources have already been created** for us:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | hello-world /    templates /      deployment.yaml      service.yaml      ingress.yaml      ...... |

We may need some of these and possibly other resources in our application, which we'll have to create ourselves as templates.

For this tutorial, we'll create deployment and a service to expose that deployment. Please note the emphasis here is not to understand Kubernetes in detail. Hence we'll keep these resources as simple as possible.

Let's edit the file *deployment.yaml* inside the *templates* directory to look like:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | apiVersion: apps/v1  kind: Deployment  metadata:    name: {{ include "hello-world.fullname" . }}    labels:      app.kubernetes.io/name: {{ include "hello-world.name" . }}      helm.sh/chart: {{ include "hello-world.chart" . }}      app.kubernetes.io/instance: {{ .Release.Name }}      app.kubernetes.io/managed-by: {{ .Release.Service }}  spec:    replicas: {{ .Values.replicaCount }}    selector:      matchLabels:        app.kubernetes.io/name: {{ include "hello-world.name" . }}        app.kubernetes.io/instance: {{ .Release.Name }}    template:      metadata:        labels:          app.kubernetes.io/name: {{ include "hello-world.name" . }}          app.kubernetes.io/instance: {{ .Release.Name }}      spec:        containers:          - name: {{ .Chart.Name }}            image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"            imagePullPolicy: {{ .Values.image.pullPolicy }}            ports:              - name: http                containerPort: 8080                protocol: TCP |

Similarly, let's edit the file *service.yaml* to look like:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | apiVersion: v1  kind: Service  metadata:    name: {{ include "hello-world.fullname" . }}    labels:      app.kubernetes.io/name: {{ include "hello-world.name" . }}      helm.sh/chart: {{ include "hello-world.chart" . }}      app.kubernetes.io/instance: {{ .Release.Name }}      app.kubernetes.io/managed-by: {{ .Release.Service }}  spec:    type: {{ .Values.service.type }}    ports:      - port: {{ .Values.service.port }}        targetPort: http        protocol: TCP        name: http    selector:      app.kubernetes.io/name: {{ include "hello-world.name" . }}      app.kubernetes.io/instance: {{ .Release.Name }} |

Now, with our knowledge of Kubernetes, these template files look quite familiar except for some oddities. **Note the liberal usage of text within double parentheses {{}}. This is what is called a template directive.**

**Helm makes use of the Go template language and extends that to something called Helm template language.** During the evaluation, every file inside the template directory is submitted to the template rendering engine. This is where the template directive injects actual values in the templates.

**7.3. Providing Values**

In the previous sub-section, we saw how to use the template directive in our templates. Now, let's understand how we can pass values to the template rendering engine. **We typically pass values through Built-in Objects in Helm.**

There are many such objects available in Helm, like Release, Values, Chart, and Files.

We can use the file *values.yaml* in our chart to pass values to the template rendering engine through the Built-in Object Values. Let's modify the *values.yaml* to look like:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | replicaCount: 1  image:    repository: "hello-world"    tag: "1.0"    pullPolicy: IfNotPresent  service:    type: NodePort    port: 80 |

However, note how these values have been accessed within templates using dots separating namespaces. We have used the image repository and tag as “hello-world” and “1.0”, this must match the docker image tag we created for our Spring Boot application.

**8. Understanding Helm Commands**

With everything done so far, we're now ready to play with our chart. Let's see what are the different commands available in Helm CLI to make this fun!

**8.1. Helm Lint**

Firstly, this is a simple command that takes the path to a chart and runs a battery of tests to ensure that the chart is well-formed:

|  |  |
| --- | --- |
| 1  2  3 | helm lint ./hello-world  ==> Linting ./hello-world  1 chart(s) linted, no failures |

**8.2 Helm Template**

Also, we have this command to render the template locally, without a Tiller Server, for quick feedback:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54 | helm template ./hello-world  ---  # Source: hello-world/templates/service.yaml  apiVersion: v1  kind: Service  metadata:    name: release-name-hello-world    labels:      app.kubernetes.io/name: hello-world      helm.sh/chart: hello-world-0.1.0      app.kubernetes.io/instance: release-name      app.kubernetes.io/managed-by: Tiller  spec:    type: NodePort    ports:      - port: 80        targetPort: http        protocol: TCP        name: http    selector:      app.kubernetes.io/name: hello-world      app.kubernetes.io/instance: release-name    ---  # Source: hello-world/templates/deployment.yaml  apiVersion: apps/v1  kind: Deployment  metadata:    name: release-name-hello-world    labels:      app.kubernetes.io/name: hello-world      helm.sh/chart: hello-world-0.1.0      app.kubernetes.io/instance: release-name      app.kubernetes.io/managed-by: Tiller  spec:    replicas: 1    selector:      matchLabels:        app.kubernetes.io/name: hello-world        app.kubernetes.io/instance: release-name    template:      metadata:        labels:          app.kubernetes.io/name: hello-world          app.kubernetes.io/instance: release-name      spec:        containers:          - name: hello-world            image: "hello-world:1.0"            imagePullPolicy: IfNotPresent            ports:              - name: http                containerPort: 8080                protocol: TCP |

**8.3. Helm Install**

Once we've verified the chart to be fine, finally, we can run this command to install the chart into the Kubernetes cluster:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | helm install --name hello-world ./hello-world  NAME:   hello-world  LAST DEPLOYED: Mon Feb 25 15:29:59 2019  NAMESPACE: default  STATUS: DEPLOYED    RESOURCES:  ==> v1/Service  NAME         TYPE      CLUSTER-IP     EXTERNAL-IP  PORT(S)       AGE  hello-world  NodePort  10.110.63.169  <none>       80:30439/TCP  1s    ==> v1/Deployment  NAME         DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE  hello-world  1        0        0           0          1s    ==> v1/Pod(related)  NAME                          READY  STATUS   RESTARTS  AGE  hello-world-7758b9cdf8-cs798  0/1    Pending  0         0s |

Finally, note that we have named the release of this chart with the flag –name. The command responds with the summary of Kubernetes resources created in the process.

**8.4. Helm Get**

Now, we would like to see which charts are installed as what release. This command lets us query the named releases:

|  |  |
| --- | --- |
| 1  2  3 | helm ls --all  NAME            REVISION        UPDATED                         STATUS          CHART               APP VERSION NAMESPACE  hello-world     1               Mon Feb 25 15:29:59 2019        DEPLOYED        hello-world-0.1.0   1.0         default |

**8.5. Helm Upgrade**

What if we have modified our chart and need to install the updated version? This command helps us to upgrade a release to a specified or current version of the chart or configuration:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | helm upgrade hello-world ./hello-world  Release "hello-world" has been upgraded. Happy Helming!  LAST DEPLOYED: Mon Feb 25 15:36:04 2019  NAMESPACE: default  STATUS: DEPLOYED    RESOURCES:  ==> v1/Service  NAME         TYPE      CLUSTER-IP     EXTERNAL-IP  PORT(S)       AGE  hello-world  NodePort  10.110.63.169  <none>       80:30439/TCP  6m5s    ==> v1/Deployment  NAME         DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE  hello-world  1        1        1           1          6m5s    ==> v1/Pod(related)  NAME                          READY  STATUS   RESTARTS  AGE  hello-world-7758b9cdf8-cs798  1/1    Running  0         6m4s |

**8.6. Helm Rollback**

It can always happen that a release went wrong and needs to be taken back. This is the command to rollback a release to the previous version:

|  |  |
| --- | --- |
| 1  2 | helm rollback hello-world 1  Rollback was a success! Happy Helming! |

**8.7. Helm Delete**

Although less likely, we may want to delete a release completely. We can use this command to delete a release from Kubernetes:

|  |  |
| --- | --- |
| 1  2 | helm delete --purge hello-world  release "hello-world" deleted |

These are only some of the commands available to work with charts and releases in Helm.

**9. Distributing Charts**

While templating is a powerful tool that Helm brings to the world of managing Kubernetes resources, it's not the only benefit of using Helm. As we saw in the previous section, Helm acts as a package manager for the Kubernetes application and makes installing, querying, upgrading, and deleting releases pretty seamless.

In addition to this, **Helm comes with commands as part of its CLI to package, publish, and fetch Kubernetes applications as charts**:

**9.1. Helm Package**

Firstly, we need to package the charts we have created to be able to distribute them. This is the command to create versioned archive files of the chart:

|  |  |
| --- | --- |
| 1  2 | helm package ./hello-world  Successfully packaged chart and saved it to: \hello-world\hello-world-0.1.0.tgz |

Note that it produces an archive on your machine that can be distributed manually or through public or private chart repositories.

**9.2. Helm Repo**

Finally, we need a mechanism to work with shared repositories to collaborate. Repo bundles a bunch of commands that we can use to add, remove, list, or index chart repositories. Let's see how we can use them.

**We can create a git repository and use that to function as our chart repository.** The only requirement is that it should have an *index.yaml* file.

We can create *index.yaml* for our chart repo:

|  |  |
| --- | --- |
| 1 | helm repo index my-repo/ --url https://<username>.github.io/my-repo |

This generates the *index.yaml* file, which we should push to the repository along with the chart archives.

After successfully creating the chart repository, subsequently, we can remotely add this repo:

|  |  |
| --- | --- |
| 1 | helm repo add my-repo [https://my](https://my/)-pages.github.io/my-repo |

Now, we should be able to install the charts from our repo directly:

|  |  |
| --- | --- |
| 1 | helm install my-repo/hello-world --name=hello-world |

There are [quite some utility commands available](https://github.com/helm/helm/tree/master/docs/helm) to work with chart repositories.

**10. Conclusion**

To sum up, in this tutorial, we discussed the core components of Helm, a package manager for Kubernetes applications. We understood options to install Helm. Furthermore, we went through creating a sample chart and templates with values.

Then, we went through multiple commands available as part of Helm CLI to manage the Kubernetes application as a Helm package.

Finally, we discussed the options for distributing Helm packages through repositories.