(/)

(https://freestar.com/?

Using Helm and Kubernetes

Last modified: November 30, 2022

by Kumar Chandrakant (https://www.baeldung.com/author/kumar-chandrakant)

DevOps (https://www.baeldung.com/category/devops)

Kubernetes (https://www.baeldung.com/tag/kubernetes)

Using **Foresight (https://www.baeldung.com/ops/foresight-understanding-ci-pipelines) to Understand our CI Pipeline**

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 awarded the graduated status by Cloud Native Computing Foundation (CNCF) (https://www.cncf.io/), which shows its growing popularity amongst Kubernetes users.

2. Background

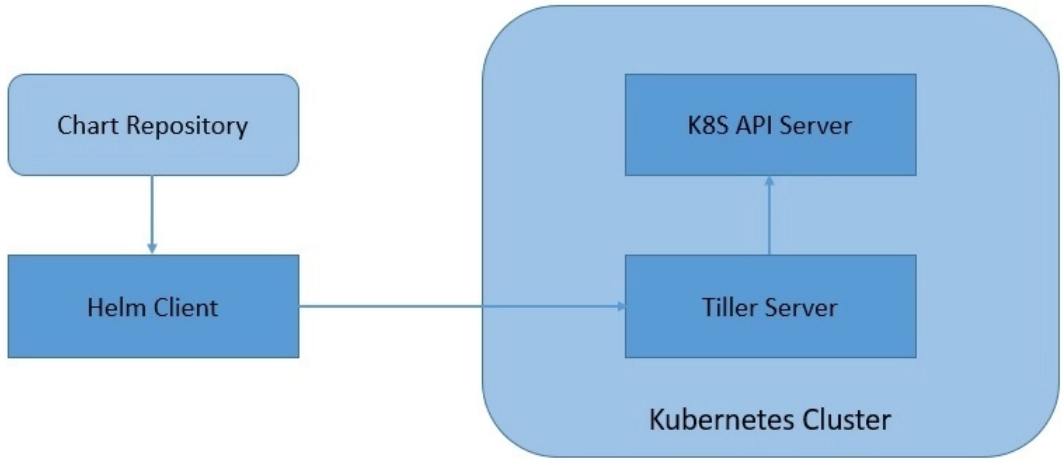
Although these terms are fairly common these days, particularly amongst those working with cloud technologies, let's go through them quickly for those unaware:

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, used by Docker Command-Line Interface (CLI). Please refer to this article for a more detailed description of Docker (/dockerizing-spring-boot-application).
3. Kubernetes (https://kubernetes.io/): **Kubernetes is a popular container orchestration program**. Although it's 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 (/kubernetes).

3. Helm Architecture

Helm has undergone a significant architecture uplift as part of Helm 3. It has some of the significant and long-awaited changes as compared to Helm 2. Apart from packing a new set of capabilities, Helm 3 also features changes in its internal plumbing. We'll examine some of these changes.

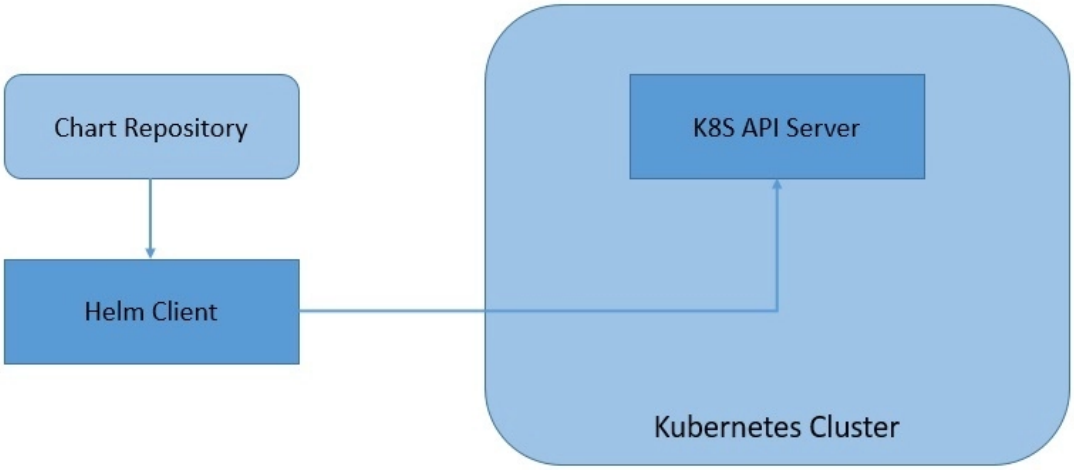
Helm 2 was primarily on a client-server architecture that comprises of a client and an in-cluster server:



(/wp-content/uploads/2019/03/Helm-2-Architecture.jpg)

- Tiller Server: **Helm manages the Kubernetes application through Tiller Server** installed within a Kubernetes 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**. It is responsible for interacting with the Tiller server to perform various operations like install, upgrade and rollback charts.

Helm 3 has **moved onto a completely client-only architecture**, where the in-cluster server has been removed:



(/wp-content/uploads/2019/03/Helm-3-Architecture.jpg)
As we can see, the client in Helm 3 works pretty much the same but **interacts directly with the Kubernetes API server** instead of the Tiller server. This move has simplified the architecture of Helm and allowed it to leverage the Kubernetes user cluster security.

4. Helm Charts, Releases, and Repositories

Helm manages Kubernetes resource packages through Charts. Charts are basically the packaging format for Helm. The chart infrastructure has also gone some changes as part of Helm 3 compared to Helm 2.

We'll see more about charts and the changes in Helm 3 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**

Helm 3 also introduced the concept of library charts. Basically, **library charts enable support for common charts** that we can use to define chart primitives or definitions. This can help to share snippets of code that we can re-use across charts.

(https://freestar.com/?

Helm **tracks an installed chart in the Kubernetes cluster using releases**. This allows us to install a single chart multiple times with different releases in a cluster. Until Helm 2, releases were stored as ConfigMaps or Secrets in the cluster under the Tiller namespace. Starting with Helm 3, releases are stored as Secrets by default in the namespace of the release directly.

Finally, we can **share charts as archives through repositories**. It is basically a location where packages charts can be stored and shared. There is a distributed community chart repository by the name Artifact Hub (https://artifacthub.io/) where we can collaborate. We can also create our own private chart repositories. We can add any number of chart repositories to work with.

5. Prerequisites

We'll need a few things to be set up beforehand 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 detail (/spring-boot-minikube).

It is often advisable to install **the most compatible version of the Kubernetes** as supported by Helm (https://helm.sh/docs/topics/version_skew/). We should also install and

configure the Kubernetes command-line tool *kubectl, enabling* us to work with our cluster efficiently.

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 (/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/intro/install/). **The quickest way to install helm on Windows is using Chocolaty (https://chocolatey.org/)**, a package manager for Windows platforms.

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

```
choco install kubernetes-helm
```

This installs the Helm client locally. This also provides us with the Helm command-line tool that we'll use to work with Helm in this tutorial.

Before proceeding further, we should **ensure that the Kubernetes cluster is running** and accessible using the *kubectl* command*:*

```
kubectl cluster-info
```

Now, until Helm 2, it was also necessary to initialize Helm. This effectively installs the Tiller server and sets up the Helm state onto a Kubernetes cluster. We could initialize Helm through the Helm CLI using the command:

```
helm init
```

But, starting with Helm 3, since there is no more Tiller server, **it's unnecessary to initialize Helm**. In fact, this command has been removed. Consequently, the Helm state is created automatically when required.

7. Developing Our First Chart

Now we are ready to develop our first Helm Chart with templates and values. We'll use the Helm CLI that was installed earlier to perform some of the common activities related to a chart.

(https://freestar.com/?

7.1. Creating a Chart

The first step, of course, would be to create a new chart with a given name:

```
helm create hello-world
```

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

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

```
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:

```
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 a deployment and service to expose that deployment. Please note that 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:

```
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:

```
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 into 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:

```
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. Managing Charts

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

(https://freestar.com/?

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:

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

The output displays the result of the linting with issues that it identifies.

8.2. Helm Template

Also, we've this command to render the template locally for quick feedback:

```
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
```

Please note that this command fakes the values that are otherwise expected to be retrieved in the cluster.

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:

```
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
```

This command also provides several options to override the values in a chart. Note that we've 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:

```
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
```

There are several sub-commands available for this command to get the extended information. These include All, Hooks, Manifest, Notes, and Values.

8.5. Helm Upgrade

What if we've 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:

```
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
```

Please note that with Helm 3, the release upgrade uses a three-way strategic merge patch. Here, it considers the old manifest, cluster live state, and new when generating a patch. Helm 2 used a two-way strategic merge patch that discarded changes applied to the cluster outside of Helm.

8.6. Helm Rollback

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

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

We can specify a specific version to roll back to or leave this argument black, in which case it rolls back to the previous version.

8.7. Helm Uninstall

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

```
helm uninstall hello-world
release "hello-world" deleted
```

It removes all of the resources associated with the last release of the chart and the release history.

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, we can also use Helm to package, publish, and fetch Kubernetes applications as chart archives. We can also use the Helm CLI for this as it offers several commands to perform these activities. As before, we'll not cover all the available commands.

9.1. Helm Package

Firstly, we need to package the charts we've created to be able to distribute them. This is the command to create a versioned archive file of the chart:

```
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 our machine that we can distribute manually or through public or private chart repositories. We also have an option to sign the chart archive.

9.2. Helm Repo

Finally, we need a mechanism to work with shared repositories to collaborate. There are several sub-commands available within this command that we can use to add, remove, update, 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:

```
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:

```
helm repo add my-repo https://my-pages.github.io/my-repo
```

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

```
helm install my-repo/hello-world --name=hello-world
```

There are quite a several commands available to work with the chart repositories.

9.3. Helm Search

Finally, we should search for a keyword within a chart that can be present on any public or private chart repositories.

```
helm search repo <KEYWORD>
```

There are sub-commands available for this command that allows us to search different locations for charts. For instance, we can search for charts in the Artifact Hub or our own repositories. Further, we can search for a keyword in the charts available in all the repositories we've configured.

10. Migration from Helm 2 to Helm 3

Since Helm has been in use for a while, it's obvious to suspect the future of Helm 2 with the significant changes as part of Helm 3. While it's advisable to start with Helm 3 if we are starting fresh, support for Helm 2 will continue in Helm 3 for the near future. Although, there are caveats, and hence will have to make necessary accommodations.

Some of the important changes to note include that Helm 3 no longer automatically generates the release name. However, we've got the necessary flag that we can use to generate the release name. Moreover, the namespaces are no longer created when a release is created. We should create the namespaces in advance.

But there are a couple of options for a project that uses Helm 2 and wishes to migrate to Helm 3. First, we can use Helm 2 and Helm 3 to manage the same cluster and slowly drain away Helm 2 releases while using Helm 3 for new releases. Alternatively, we can decide to manage Helm 2 releases using Helm 3. While this can be tricky, Helm provides a plugin to handle this type of migration (https://github.com/helm/helm-2to3).

11. Conclusion

To sum up, in this tutorial, we discussed the core components of Helm, a package manager for Kubernetes applications. We understood the 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. In the process, we saw the changes that have been done as part of Helm 3 compared to Helm 2.

Get started with Spring 5 and Spring Boot 2, through the *Learn Spring* course:

>> CHECK OUT THE COURSE (/ls-course-end)

Learning to build your API
with Spring?

Download the E-book (/rest-api-spring-guide)

Comments are closed on this article!

COURSES

ALL COURSES (/ALL-COURSES)
ALL BULK COURSES (/ALL-BULK-COURSES)
ALL BULK TEAM COURSES (/ALL-BULK-TEAM-COURSES)
THE COURSES PLATFORM (HTTPS://COURSES.BAELDUNG.COM)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
APACHE HTTPCLIENT TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)
SPRING REACTIVE TUTORIALS (/SPRING-REACTIVE-GUIDE)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE FULL ARCHIVE (/FULL_ARCHIVE)
EDITORS (/EDITORS)
JOBS (/TAG/ACTIVE-JOB/)

OUR PARTNERS (/PARTNERS)
PARTNER WITH BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)