# Helm Lab

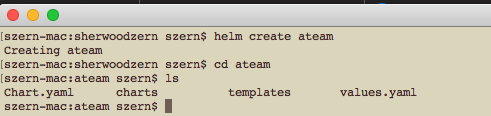
Create a simple chart called “aname”

$ helm create “aname”

You will see the message:

*Creating “aname”*

When the chart creation has been completed you will have a directory with the name of <aname> with a number of sub-directories. As an example:



The create operation creates 2 directories; charts and templates, and 2 files; values.yaml and Chart.yaml.

$ *ls ./templates*

A view of the ateam/templates directory shows the following:



* NOTES.txt: The “help text” for your chart. This will be displayed to your users when they run helm intall.
* deployment.yaml: A basic manifest for creating a Kubernetes deployment.
* service.yaml: A basic manifest for creating a service endpoint for your deployment.
* \_helpers.tpl: A place to put template helpers that you can re-use throughout the chart

In order to demonstrate how Helm works we will remove all of the files in the templates subdirectory. Removing these files is not normally done, but since we will not actually be deploying a deployment or a service we will do so just to demonstrate how Helm works.

$ *rm -rf ./templates/\*.\**



Let’s begin by creating a simple YAML file to create a ConfigMap.

apiVersion: v1

kind: ConfigMap

metadata:

name: ateam-configmap

data:

myvalue: "Hello World"

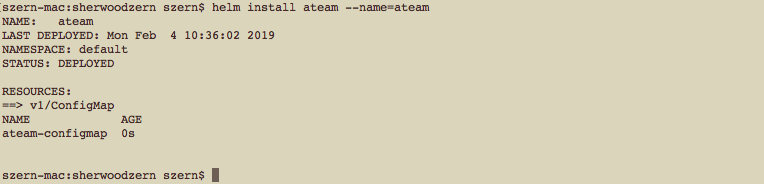
Save the file as configmap.yaml in the templates directory. IMPORTANT: When creating the file please watch the spacing. There are two spaces prior to “name” and “myvalue”. If there are too many or not enough spaces then the YAML validator will throw an error. In addition, the metadata.name value must be all lower-case

When Tiller (Helm server) reads this template, it will simply send it to Kubernetes as-is.

Let’s install our basic ConfigMap Helm chart.

$ *helm install ateam –name=ateam (2 dashes before name)*

If you do not specify a –name then Helm will generate a default name.



We can see that the ConfigMap was created. Using Helm, we can retrieve the release and see the actual template that was loaded.

Let’s expand our example to demonstrate the real power of Helm. First we’ll remove the ConfigMap that we recently deployed.

$ *helm delete –purge ateam*

Hard-coding the *name:* into a resource is usually considered to be bad practice. Names should be unique to a release.

**TIP:** The *name:* field is limited to 63 characters because of limitations to the DNS system. For that reason, release names are limited to 53 characters.

Alter the configmap.yaml as such:

apiVersion: v1

kind: ConfigMap

metadata:

name: {{ .Release.Name }}-configmap

data:

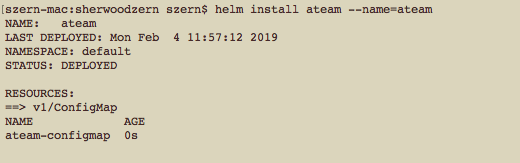
myvalue: "Hello World"

A template directive is enclosed in “{{ and }}” blocks. There are two spaces after the first set of {{ and two spaces before }}.

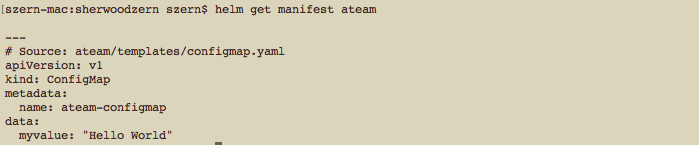
The template directive {{ .Release.Name }} injects the release name into the template. The values that are passed into a template can be thought of a *namespaced objects*, where a “.” Separates each namespaced element.

The *Release* object is one of the built-in objects for Helm.

$ *helm install ateam –name=ateam*



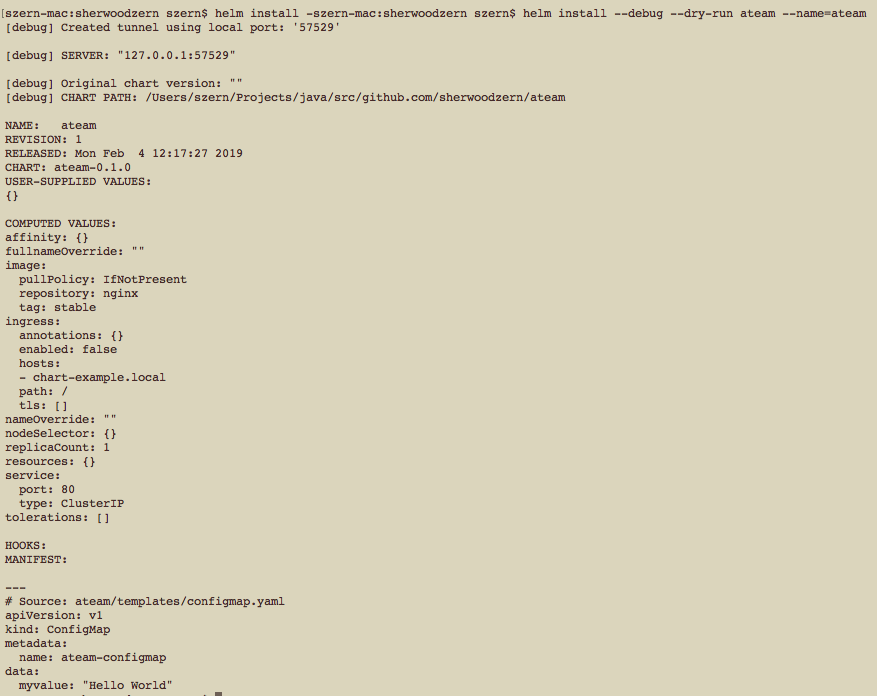
$ *helm get manifest ateam*

**

At this point, we’ve seen templates at their most basic: YAML files that have template directives embedded in {{ and }}. In the next part, we’ll take a deeper look into templates. But before moving on, there’s one quick trick that can make building templates faster: When you want to test the template rendering, but not actually install anything, you can use

$ *helm install --debug --dry-run ./ateam*

This will send the chart to the Tiller server, which will render the templates. But instead of installing the chart, it will return the rendered template to you so you can see the output:



Using –dry-run will make it easier to test your code, but it won’t ensure that Kubernetes itself will accept the templates you generate. It’s best not to assume that your chart will install just because –dry-run works.

Helm has four built-in Objects. These objects are Release, Values, Files, and Capabilities. You previously saw the use of the Release object. We’re not going to cover all of these objects and the inner objects within each of these top-level built-in Objects. I will leave the investigation of these top-level objects and the inner objects to the attendee. However, we will cover the top-level Object Values. The Values top-level object is one of those objects that you will use most frequently.

The Values object provides access to values passed into the chart. The contents come from four sources:

* The *values.yaml* file in the chart
* If this is a subchart, the *values.yaml* file of the parent chart
* A values file is passed into the *helm install* or *helm upgrade* with the -f flag
* Individual parameters passed with –set (2 dashes). As an example, *helm install –set key=value ateam*

The list above is in order of specificity: values.yaml is the default, which can be overridden by a parent chart’s values.yaml, which can in turn be overridden by a user-supplied values file, which can in turn by overridden by –set parameters.

Let’s make some changes to our configmap.yaml and values.yaml file. We can then redeploy to see how our changes to the files have impacted the deployment.

**configmap.yaml file**

apiVersion: v1

kind: ConfigMap

metadata:

name: {{ .Release.Name }}-configmap

data:

myvalue: "Hello World"

animal: {{ .Values.animals.favorite }}

**values.yaml file**

# Default values for ateam.

# This is a YAML-formatted file.

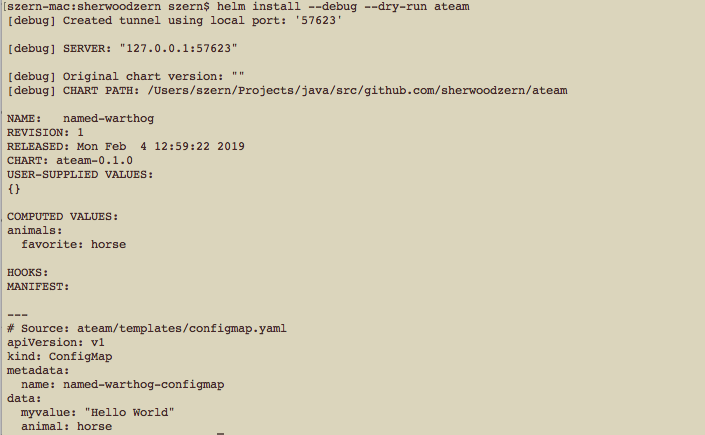
# Declare variables to be passed into your templates.

animals:

favorite: horse

Let’s do a dry-run so you can see how the ConfigMap would be rendered with the values.yaml file being used.

$ *helm install –debug –dry-run ateam*

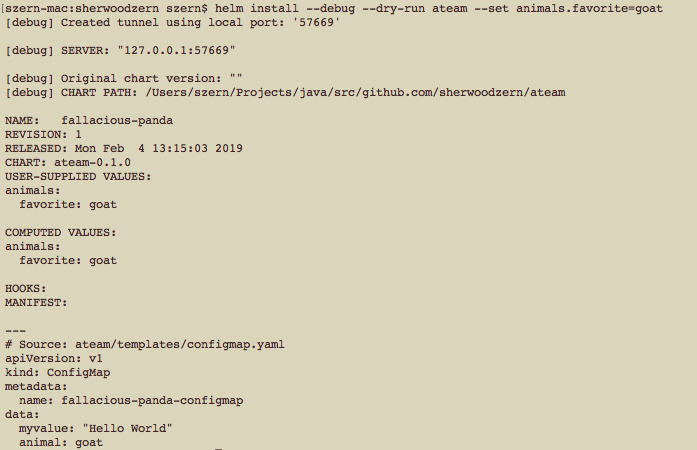


Let’s review what happened.

1. Since we did not pass in the –name parameter Helm generated its own name. In my case it is: “named-warthog”. Your name will be different and may be even crazier.
2. Notice in the data element that the key of animal has a value of horse. This value was taken from the values.yaml file.

We can also override the value in the values.yaml file by providing a value on the command line.

$ *helm install –debug –dry-run ateam –set animals.favorite=goat*



You can see that the passed in parameter overrode the values.yaml file and the name generated by Helm was auto generated: fallacious-panda.

Helm provides functions that can be used in your files. As an example:

From the configmap.yaml file we can do this. *animal: {{ .Values.animals.favorite | quote }}*

Doing this will place quotes around the resulting value, “horse”

You can also specify a default value in case a value is not provided:

*{{ .Values.animals.favorite. }} | default “chicken” | quote }}*

Helm provides many other functions, but we will not discuss them here. This is another exercise left to the student.

# Flow Control

Control structures provide you with the ability to control the flow of a template’s generation. Helm’s template language provides the following control structures:

* If/else: for creating conditional blocks
* with: to specify a scope
* range: which provides a “for-each”-style loop

Let’s do a simple flow control with the if action.

Values.yaml file

# Default values for ateam.

# This is a YAML-formatted file.

# Declare variables to be passed into your templates.

animals:

favorite: goat

recentbabies: "3"

farmer: poppy

configmap.yaml file

apiVersion: v1

kind: ConfigMap

metadata:

name: {{ .Release.Name }}-configmap

data:

myvalue: "Hello World"

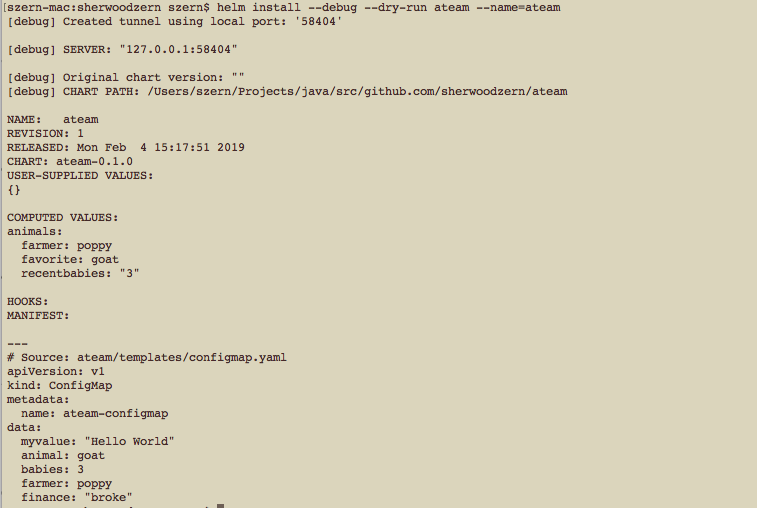
animal: {{ .Values.animals.favorite }}

babies: {{ .Values.animals.recentbabies }}

farmer: {{ .Values.animals.farmer }}

{{ if and (eq .Values.animals.favorite "goat") (gt .Values.animals.recentbabies "2") }}finance: "broke" {{ else }} {{ end }}

As you can see below the structure is evaluated and the configmap yaml file is created. It is then passed onto Kubernetes. Let’s take a look at some “gotchas” when building our Helm charts.



configmap.yaml

. . . .

. . . .

{{ if and (eq .Values.animals.favorite "goat") (gt .Values.animals.recentbabies "2") }}

finance: "broke"

{{ else }}

finance: stable

{{ end }}

Break your YAML file into multiple lines to make it more readable. Also, indent so that the if/else blocks are more readable. Once you have done this do another Helm install as a dry-run.

* What was the result? Are you able to deploy the Helm chart?

I’m sure you noticed that the template engine rejected the final YAML file. What Happened? The generated YAML is incorrect because of the whitespace. Let’s fix this.

configmap.yaml

. . . .

. . . .

{{ if and (eq .Values.animals.favorite "goat") (gt .Values.animals.recentbabies "2") }}

finance: "broke"

{{ else }}

finance: stable

{{ end }}

Remove the indentations.

* Does the resulting YAML now look correct?
* What’s the difference this time?

You probably noticed a few empty lines in your YAML. When the template engine runs, it removes the contents inside of {{ and }}, but it leaves the remaining whitespace exactly as is.

YAML ascribes meaning to whitespace, so managing the whitespace becomes pretty important. Fortunately, Helm templates have a few tools to help.

First, the curly brace syntax of template declarations can be modified with special characters to tell the template engine to chomp whitespace. {{- (with the dash and space added) indicates that whitespace should be chomped left, while -}} means whitespace to the right should be consumed. Be careful! Newlines are whitespace.

configmap.yaml

. . . .

. . . .

{{- if and (eq .Values.animals.favorite "goat") (gt .Values.animals.recentbabies "2") }}

finance: "broke"

{{- else }}

finance: stable

{{- end }}

* How did applying these changes help?

# Modifying Scope using *with*

Scopes can be changed using the *with* operation. It allows you to set the current scope to a particular object. You saw in our previous examples how we have been using {{ .Values.animals. }}. You had to specify the entire path to access the actual values. You can set the scope to a specific object.

configmap.yaml

. . . .

. . . .

{{- with .Values.animals }}

animal: {{ .favorite … }}

For this exercise do the following:

* Scope the objects to animals
* Set the babies and farmer and finance within the scope
* Set the finance to stable if the favorite animal is a goat and the number of babies is less than 3. Set the finance to broke if the favorite animal is a goat the number of babies is greater than 2
* Add the Release.Name within the scope.
* Do a dry-run to validate the generated YAML

# Looping with the *range action*

Helm’s template language provides a way to iterate through a collection. The *range* action allows you to loop through a collection.

In the values.yaml file let’s add a collection:

. . . .

. . . .

. . . .

eaters:

- horses

- goats

- chickens

- grandchildren

In order to do the looping within the YAML file use the range action. The format of the action is as follows:

<key> : |-

{{- range <Object.key> }}

- {{ . | title | <function> }}

{{- end }}

For this exercise:

* Loop through your list that you added to your values.yaml file
* Use the function in cofigmap.yaml file to place quotes around each list element
* Use the Helm –dry-run to make sure everything is built correctly.

The <key>: |- is declaring a multi-line string. The list of eaters is actually not a YAML list. It’s a big string. For our testing purposes, we did this because data in ConfigMaps data is composed of key/value pairs, where both the key and the value are simple strings.

# Summary

You should have a decent understanding as to how Helm charts work. What we did in this lab is to gain an understanding of some of the basic pieces of functionality of Helm. There is more to Helm, but we don’t have the time to cover all of the aspects of Helm.

When we move to deploying Istio we will take a look at some of the more complex use of Helm charts. Reviewing some complex Helm charts can provide you some good ideas to how best to leverage Helm charts.