**Getting Started with a Chart Template**

In this section of the guide, we’ll create a chart and then add a first template. The chart we created here will be used throughout the rest of the guide.

To get going, let’s take a brief look at a Helm chart.

**CHARTS**

As described in the [Charts Guide](https://helm.sh/docs/chart_template_guide/#../charts), Helm charts are structured like this:

mychart/

Chart.yaml

values.yaml

charts/

templates/

...

The templates/ directory is for template files. When Tiller evaluates a chart, it will send all of the files in the templates/ directory through the template rendering engine. Tiller then collects the results of those templates and sends them on to Kubernetes.

The values.yaml file is also important to templates. This file contains the *default values* for a chart. These values may be overridden by users during helm install or helm upgrade.

The Chart.yaml file contains a description of the chart. You can access it from within a template. The charts/ directory *may* contain other charts (which we call *subcharts*). Later in this guide we will see how those work when it comes to template rendering.

**A STARTER CHART**

For this guide, we’ll create a simple chart called mychart, and then we’ll create some templates inside of the chart.

$ helm create mychart

Creating mychart

From here on, we’ll be working in the mychart directory.

**A Quick Glimpse of**mychart/templates/

If you take a look at the mychart/templates/ directory, you’ll notice a few files already there.

* NOTES.txt: The “help text” for your chart. This will be displayed to your users when they run helm install.
* deployment.yaml: A basic manifest for creating a Kubernetes [deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/)
* service.yaml: A basic manifest for creating a [service endpoint](https://kubernetes.io/docs/concepts/services-networking/service/) for your deployment
* \_helpers.tpl: A place to put template helpers that you can re-use throughout the chart

And what we’re going to do is… *remove them all!* That way we can work through our tutorial from scratch. We’ll actually create our own NOTES.txt and \_helpers.tpl as we go.

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

When you’re writing production grade charts, having basic versions of these charts can be really useful. So in your day-to-day chart authoring, you probably won’t want to remove them.

**A FIRST TEMPLATE**

The first template we are going to create will be a ConfigMap. In Kubernetes, a ConfigMap is simply a container for storing configuration data. Other things, like pods, can access the data in a ConfigMap.

Because ConfigMaps are basic resources, they make a great starting point for us.

Let’s begin by creating a file called mychart/templates/configmap.yaml:

apiVersion: v1

kind: ConfigMap

metadata:

name: mychart-configmap

data:

myvalue: "Hello World"

**TIP:** Template names do not follow a rigid naming pattern. However, we recommend using the suffix .yaml for YAML files and .tpl for helpers.

The YAML file above is a bare-bones ConfigMap, having the minimal necessary fields. In virtue of the fact that this file is in the templates/ directory, it will be sent through the template engine.

It is just fine to put a plain YAML file like this in the templates/ directory. When Tiller reads this template, it will simply send it to Kubernetes as-is.

With this simple template, we now have an installable chart. And we can install it like this:

$ helm install ./mychart

NAME: full-coral

LAST DEPLOYED: Tue Nov 1 17:36:01 2016

NAMESPACE: default

STATUS: DEPLOYED

RESOURCES:

==> v1/ConfigMap

NAME DATA AGE

mychart-configmap 1 1m

In the output above, we can see that our ConfigMap was created. Using Helm, we can retrieve the release and see the actual template that was loaded.

$ helm get manifest full-coral

---

# Source: mychart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: mychart-configmap

data:

myvalue: "Hello World"

The helm get manifest command takes a release name (full-coral) and prints out all of the Kubernetes resources that were uploaded to the server. Each file begins with --- to indicate the start of a YAML document, and then is followed by an automatically generated comment line that tells us what template file generated this YAML document.

From there on, we can see that the YAML data is exactly what we put in our configmap.yaml file.

Now we can delete our release: helm delete full-coral.

**Adding a Simple Template Call**

Hard-coding the name: into a resource is usually considered to be bad practice. Names should be unique to a release. So we might want to generate a name field by inserting the release name.

**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. Kubernetes 1.3 and earlier limited to only 24 characters (thus 14 character names).

Let’s alter configmap.yaml accordingly.

apiVersion: v1

kind: ConfigMap

metadata:

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

data:

myvalue: "Hello World"

The big change comes in the value of the name: field, which is now {{ .Release.Name }}-configmap.

A template directive is enclosed in {{ and }} blocks.

The template directive {{ .Release.Name }} injects the release name into the template. The values that are passed into a template can be thought of as *namespaced objects*, where a dot (.) separates each namespaced element.

The leading dot before Release indicates that we start with the top-most namespace for this scope (we’ll talk about scope in a bit). So we could read .Release.Name as “start at the top namespace, find the Release object, then look inside of it for an object called Name”.

The Release object is one of the built-in objects for Helm, and we’ll cover it in more depth later. But for now, it is sufficient to say that this will display the release name that Tiller assigns to our release.

Now when we install our resource, we’ll immediately see the result of using this template directive:

$ helm install ./mychart

NAME: clunky-serval

LAST DEPLOYED: Tue Nov 1 17:45:37 2016

NAMESPACE: default

STATUS: DEPLOYED

RESOURCES:

==> v1/ConfigMap

NAME DATA AGE

clunky-serval-configmap 1 1m

Note that in the RESOURCES section, the name we see there is clunky-serval-configmap instead of mychart-configmap.

You can run helm get manifest clunky-serval to see the entire generated YAML.

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 ./mychart --debug --dry-run. 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:

$ helm install ./mychart --debug --dry-run

SERVER: "localhost:44134"

CHART PATH: /Users/mattbutcher/Code/Go/src/k8s.io/helm/\_scratch/mychart

NAME: goodly-guppy

TARGET NAMESPACE: default

CHART: mychart 0.1.0

MANIFEST:

---

# Source: mychart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: goodly-guppy-configmap

data:

myvalue: "Hello World"

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.

In the next few sections, we’ll take the basic chart we defined here and explore the Helm template language in detail. And we’ll get started with built-in objects.

**Built-in Objects**

Objects are passed into a template from the template engine. And your code can pass objects around (we’ll see examples when we look at the with and range statements). There are even a few ways to create new objects within your templates, like with the list function we’ll see later.

Objects can be simple, and have just one value. Or they can contain other objects or functions. For example. the Release object contains several objects (like Release.Name) and the Files object has a few functions.

In the previous section, we use {{.Release.Name}} to insert the name of a release into a template. Release is one of the top-level objects that you can access in your templates.

* Release: This object describes the release itself. It has several objects inside of it:
  + Release.Name: The release name
  + Release.Time: The time of the release
  + Release.Namespace: The namespace to be released into (if the manifest doesn’t override)
  + Release.Service: The name of the releasing service (always Tiller).
  + Release.Revision: The revision number of this release. It begins at 1 and is incremented for each helm upgrade.
  + Release.IsUpgrade: This is set to true if the current operation is an upgrade or rollback.
  + Release.IsInstall: This is set to true if the current operation is an install.
* Values: Values passed into the template from the values.yaml file and from user-supplied files. By default, Values is empty.
* Chart: The contents of the Chart.yaml file. Any data in Chart.yaml will be accessible here. For example {{.Chart.Name}}-{{.Chart.Version}} will print out the mychart-0.1.0.
  + The available fields are listed in the [Charts Guide](https://github.com/helm/helm/blob/master/docs/charts.md#the-chartyaml-file)
* Files: This provides access to all non-special files in a chart. While you cannot use it to access templates, you can use it to access other files in the chart. See the section *Accessing Files* for more.
  + Files.Get is a function for getting a file by name (.Files.Get config.ini)
  + Files.GetBytes is a function for getting the contents of a file as an array of bytes instead of as a string. This is useful for things like images.
* Capabilities: This provides information about what capabilities the Kubernetes cluster supports.
  + Capabilities.APIVersions is a set of versions.
  + Capabilities.APIVersions.Has $version indicates whether a version (e.g., batch/v1) or resource (e.g., apps/v1/Deployment) is available on the cluster. Note, resources were not available before Helm v2.15.
  + Capabilities.KubeVersion provides a way to look up the Kubernetes version. It has the following values: Major, Minor, GitVersion, GitCommit, GitTreeState, BuildDate, GoVersion, Compiler, and Platform.
  + Capabilities.TillerVersion provides a way to look up the Tiller version. It has the following values: SemVer, GitCommit, and GitTreeState.
* Template: Contains information about the current template that is being executed
  + Name: A namespaced filepath to the current template (e.g. mychart/templates/mytemplate.yaml)
  + BasePath: The namespaced path to the templates directory of the current chart (e.g. mychart/templates).

The values are available to any top-level template. As we will see later, this does not necessarily mean that they will be available *everywhere*.

The built-in values always begin with a capital letter. This is in keeping with Go’s naming convention. When you create your own names, you are free to use a convention that suits your team. Some teams, like the [Helm Charts](https://github.com/helm/charts) team, choose to use only initial lower case letters in order to distinguish local names from those built-in. In this guide, we follow that convention.

**Values Files**

In the previous section we looked at the built-in objects that Helm templates offer. One of these built-in objects is Values. This object provides access to values passed into the chart. Its contents come from four sources:

* The values.yaml file in the chart
* If this is a subchart, the values.yaml file of a parent chart
* A values file is passed into helm install or helm upgrade with the -f flag (helm install -f myvals.yaml ./mychart)
* Individual parameters passed with --set (such as helm install --set foo=bar ./mychart)

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 be overridden by --set parameters.

Values files are plain YAML files. Let’s edit mychart/values.yaml and then edit our ConfigMap template.

Removing the defaults in values.yaml, we’ll set just one parameter:

favoriteDrink: coffee

Now we can use this inside of a template:

apiVersion: v1

kind: ConfigMap

metadata:

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

data:

myvalue: "Hello World"

drink: {{ .Values.favoriteDrink }}

Notice on the last line we access favoriteDrink as an attribute of Values: {{ .Values.favoriteDrink }}.

Let’s see how this renders.

$ helm install --dry-run --debug ./mychart

SERVER: "localhost:44134"

CHART PATH: /Users/mattbutcher/Code/Go/src/k8s.io/helm/\_scratch/mychart

NAME: geared-marsupi

TARGET NAMESPACE: default

CHART: mychart 0.1.0

MANIFEST:

---

# Source: mychart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: geared-marsupi-configmap

data:

myvalue: "Hello World"

drink: coffee

Because favoriteDrink is set in the default values.yaml file to coffee, that’s the value displayed in the template. We can easily override that by adding a --set flag in our call to helm install:

helm install --dry-run --debug --set favoriteDrink=slurm ./mychart

SERVER: "localhost:44134"

CHART PATH: /Users/mattbutcher/Code/Go/src/k8s.io/helm/\_scratch/mychart

NAME: solid-vulture

TARGET NAMESPACE: default

CHART: mychart 0.1.0

MANIFEST:

---

# Source: mychart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: solid-vulture-configmap

data:

myvalue: "Hello World"

drink: slurm

Since --set has a higher precedence than the default values.yaml file, our template generates drink: slurm.

Values files can contain more structured content, too. For example, we could create a favorite section in our values.yaml file, and then add several keys there:

favorite:

drink: coffee

food: pizza

Now we would have to modify the template slightly:

apiVersion: v1

kind: ConfigMap

metadata:

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

data:

myvalue: "Hello World"

drink: {{ .Values.favorite.drink }}

food: {{ .Values.favorite.food }}

While structuring data this way is possible, the recommendation is that you keep your values trees shallow, favoring flatness. When we look at assigning values to subcharts, we’ll see how values are named using a tree structure.

**DELETING A DEFAULT KEY**

If you need to delete a key from the default values, you may override the value of the key to be null, in which case Helm will remove the key from the overridden values merge.

For example, the stable Drupal chart allows configuring the liveness probe, in case you configure a custom image. Here are the default values:

livenessProbe:

httpGet:

path: /user/login

port: http

initialDelaySeconds: 120

If you try to override the livenessProbe handler to exec instead of httpGet using --set livenessProbe.exec.command=[cat,docroot/CHANGELOG.txt], Helm will coalesce the default and overridden keys together, resulting in the following YAML:

livenessProbe:

httpGet:

path: /user/login

port: http

exec:

command:

- cat

- docroot/CHANGELOG.txt

initialDelaySeconds: 120

However, Kubernetes would then fail because you can not declare more than one livenessProbe handler. To overcome this, you may instruct Helm to delete the livenessProbe.httpGet by setting it to null:

helm install stable/drupal --set image=my-registry/drupal:0.1.0 --set livenessProbe.exec.command=[cat,docroot/CHANGELOG.txt] --set livenessProbe.httpGet=null

At this point, we’ve seen several built-in objects, and used them to inject information into a template. Now we will take a look at another aspect of the template engine: functions and pipelines.

**Template Functions and Pipelines**

So far, we’ve seen how to place information into a template. But that information is placed into the template unmodified. Sometimes we want to transform the supplied data in a way that makes it more usable to us.

Let’s start with a best practice: When injecting strings from the .Values object into the template, we ought to quote these strings. We can do that by calling the quote function in the template directive:

apiVersion: v1

kind: ConfigMap

metadata:

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

data:

myvalue: "Hello World"

drink: {{ quote .Values.favorite.drink }}

food: {{ quote .Values.favorite.food }}

Template functions follow the syntax functionName arg1 arg2.... In the snippet above, quote .Values.favorite.drink calls the quote function and passes it a single argument.

Helm has over 60 available functions. Some of them are defined by the [Go template language](https://godoc.org/text/template) itself. Most of the others are part of the [Sprig template library](https://godoc.org/github.com/Masterminds/sprig). We’ll see many of them as we progress through the examples.

While we talk about the “Helm template language” as if it is Helm-specific, it is actually a combination of the Go template language, some extra functions, and a variety of wrappers to expose certain objects to the templates. Many resources on Go templates may be helpful as you learn about templating.

**PIPELINES**

One of the powerful features of the template language is its concept of *pipelines*. Drawing on a concept from UNIX, pipelines are a tool for chaining together a series of template commands to compactly express a series of transformations. In other words, pipelines are an efficient way of getting several things done in sequence. Let’s rewrite the above example using a pipeline.

apiVersion: v1

kind: ConfigMap

metadata:

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

data:

myvalue: "Hello World"

drink: {{ .Values.favorite.drink | quote }}

food: {{ .Values.favorite.food | quote }}

In this example, instead of calling quote ARGUMENT, we inverted the order. We “sent” the argument to the function using a pipeline (|): .Values.favorite.drink | quote. Using pipelines, we can chain several functions together:

apiVersion: v1

kind: ConfigMap

metadata:

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

data:

myvalue: "Hello World"

drink: {{ .Values.favorite.drink | quote }}

food: {{ .Values.favorite.food | upper | quote }}

Inverting the order is a common practice in templates. You will see .val | quote more often than quote .val. Either practice is fine.

When evaluated, that template will produce this:

# Source: mychart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: trendsetting-p-configmap

data:

myvalue: "Hello World"

drink: "coffee"

food: "PIZZA"

Note that our original pizza has now been transformed to "PIZZA".

When pipelining arguments like this, the result of the first evaluation (.Values.favorite.drink) is sent as the *last argument to the function*. We can modify the drink example above to illustrate with a function that takes two arguments: repeat COUNT STRING:

apiVersion: v1

kind: ConfigMap

metadata:

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

data:

myvalue: "Hello World"

drink: {{ .Values.favorite.drink | repeat 5 | quote }}

food: {{ .Values.favorite.food | upper | quote }}

The repeat function will echo the given string the given number of times, so we will get this for output:

# Source: mychart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: melting-porcup-configmap

data:

myvalue: "Hello World"

drink: "coffeecoffeecoffeecoffeecoffee"

food: "PIZZA"

**USING THE**DEFAULT**FUNCTION**

One function frequently used in templates is the default function: default DEFAULT\_VALUE GIVEN\_VALUE. This function allows you to specify a default value inside of the template, in case the value is omitted. Let’s use it to modify the drink example above:

drink: {{ .Values.favorite.drink | default "tea" | quote }}

If we run this as normal, we’ll get our coffee:

# Source: mychart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: virtuous-mink-configmap

data:

myvalue: "Hello World"

drink: "coffee"

food: "PIZZA"

Now, we will remove the favorite drink setting from values.yaml:

favorite:

#drink: coffee

food: pizza

Now re-running helm install --dry-run --debug ./mychart will produce this YAML:

# Source: mychart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: fair-worm-configmap

data:

myvalue: "Hello World"

drink: "tea"

food: "PIZZA"

In an actual chart, all static default values should live in the values.yaml, and should not be repeated using the default command (otherwise they would be redundant). However, the default command is perfect for computed values, which can not be declared inside values.yaml. For example:

drink: {{ .Values.favorite.drink | default (printf "%s-tea" (include "fullname" .)) }}

In some places, an if conditional guard may be better suited than default. We’ll see those in the next section.

Template functions and pipelines are a powerful way to transform information and then insert it into your YAML. But sometimes it’s necessary to add some template logic that is a little more sophisticated than just inserting a string. In the next section we will look at the control structures provided by the template language.

**OPERATORS ARE FUNCTIONS**

Operators are implemented as functions that return a boolean value. To use eq, ne, lt, gt, and, or, not etcetera place the operator at the front of the statement followed by its parameters just as you would a function. To chain multiple operations together, separate individual functions by surrounding them with parentheses.

{{/\* include the body of this if statement when the variable .Values.fooString exists and is set to "foo" \*/}}

{{ if and .Values.fooString (eq .Values.fooString "foo") }}

{{ ... }}

{{ end }}

{{/\* include the body of this if statement when the variable .Values.anUnsetVariable is set or .values.aSetVariable is not set \*/}}

{{ if or .Values.anUnsetVariable (not .Values.aSetVariable) }}

{{ ... }}

{{ end }}

Now we can turn from functions and pipelines to flow control with conditions, loops, and scope modifiers.

**Flow Control**

Control structures (called “actions” in template parlance) provide you, the template author, 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

In addition to these, it provides a few actions for declaring and using named template segments:

* define declares a new named template inside of your template
* template imports a named template
* block declares a special kind of fillable template area

In this section, we’ll talk about if, with, and range. The others are covered in the “Named Templates” section later in this guide.

**IF/ELSE**

The first control structure we’ll look at is for conditionally including blocks of text in a template. This is the if/else block.

The basic structure for a conditional looks like this:

{{ if PIPELINE }}

# Do something

{{ else if OTHER PIPELINE }}

# Do something else

{{ else }}

# Default case

{{ end }}

Notice that we’re now talking about *pipelines* instead of values. The reason for this is to make it clear that control structures can execute an entire pipeline, not just evaluate a value.

A pipeline is evaluated as *false* if the value is:

* a boolean false
* a numeric zero
* an empty string
* a nil (empty or null)
* an empty collection (map, slice, tuple, dict, array)

In any other case, the condition is evaluated to *true* and the pipeline is executed.

Let’s add a simple conditional to our ConfigMap. We’ll add another setting if the drink is set to coffee:

apiVersion: v1

kind: ConfigMap

metadata:

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

data:

myvalue: "Hello World"

drink: {{ .Values.favorite.drink | default "tea" | quote }}

food: {{ .Values.favorite.food | upper | quote }}

{{ if and .Values.favorite.drink (eq .Values.favorite.drink "coffee") }}mug: true{{ end }}

Note that .Values.favorite.drink must be defined or else it will throw an error when comparing it to “coffee”. Since we commented out drink: coffee in our last example, the output should not include a mug: true flag. But if we add that line back into our values.yaml file, the output should look like this:

# Source: mychart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: eyewitness-elk-configmap

data:

myvalue: "Hello World"

drink: "coffee"

food: "PIZZA"

mug: true

**CONTROLLING WHITESPACE**

While we’re looking at conditionals, we should take a quick look at the way whitespace is controlled in templates. Let’s take the previous example and format it to be a little easier to read:

apiVersion: v1

kind: ConfigMap

metadata:

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

data:

myvalue: "Hello World"

drink: {{ .Values.favorite.drink | default "tea" | quote }}

food: {{ .Values.favorite.food | upper | quote }}

{{if eq .Values.favorite.drink "coffee"}}

mug: true

{{end}}

Initially, this looks good. But if we run it through the template engine, we’ll get an unfortunate result:

$ helm install --dry-run --debug ./mychart

SERVER: "localhost:44134"

CHART PATH: /Users/mattbutcher/Code/Go/src/k8s.io/helm/\_scratch/mychart

Error: YAML parse error on mychart/templates/configmap.yaml: error converting YAML to JSON: yaml: line 9: did not find expected key

What happened? We generated incorrect YAML because of the whitespacing above.

# Source: mychart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: eyewitness-elk-configmap

data:

myvalue: "Hello World"

drink: "coffee"

food: "PIZZA"

mug: true

mug is incorrectly indented. Let’s simply out-dent that one line, and re-run:

apiVersion: v1

kind: ConfigMap

metadata:

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

data:

myvalue: "Hello World"

drink: {{ .Values.favorite.drink | default "tea" | quote }}

food: {{ .Values.favorite.food | upper | quote }}

{{if eq .Values.favorite.drink "coffee"}}

mug: true

{{end}}

When we sent that, we’ll get YAML that is valid, but still looks a little funny:

# Source: mychart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: telling-chimp-configmap

data:

myvalue: "Hello World"

drink: "coffee"

food: "PIZZA"

mug: true

Notice that we received a few empty lines in our YAML. Why? 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!*

Make sure there is a space between the - and the rest of your directive. {{- 3 }} means “trim left whitespace and print 3” while {{-3}} means “print -3”.

Using this syntax, we can modify our template to get rid of those new lines:

apiVersion: v1

kind: ConfigMap

metadata:

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

data:

myvalue: "Hello World"

drink: {{ .Values.favorite.drink | default "tea" | quote }}

food: {{ .Values.favorite.food | upper | quote }}

{{- if eq .Values.favorite.drink "coffee"}}

mug: true

{{- end}}

Just for the sake of making this point clear, let’s adjust the above, and substitute an \* for each whitespace that will be deleted following this rule. an \* at the end of the line indicates a newline character that would be removed

apiVersion: v1

kind: ConfigMap

metadata:

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

data:

myvalue: "Hello World"

drink: {{ .Values.favorite.drink | default "tea" | quote }}

food: {{ .Values.favorite.food | upper | quote }}\*

\*\*{{- if eq .Values.favorite.drink "coffee"}}

mug: true\*

\*\*{{- end}}

Keeping that in mind, we can run our template through Helm and see the result:

# Source: mychart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: clunky-cat-configmap

data:

myvalue: "Hello World"

drink: "coffee"

food: "PIZZA"

mug: true

Be careful with the chomping modifiers. It is easy to accidentally do things like this:

food: {{ .Values.favorite.food | upper | quote }}

{{- if eq .Values.favorite.drink "coffee" -}}

mug: true

{{- end -}}

That will produce food: "PIZZA"mug:true because it consumed newlines on both sides.

For the details on whitespace control in templates, see the [Official Go template documentation](https://godoc.org/text/template)

Finally, sometimes it’s easier to tell the template system how to indent for you instead of trying to master the spacing of template directives. For that reason, you may sometimes find it useful to use the indent function ({{indent 2 "mug:true"}}).

**MODIFYING SCOPE USING**WITH

The next control structure to look at is the with action. This controls variable scoping. Recall that . is a reference to *the current scope*. So .Values tells the template to find the Values object in the current scope.

The syntax for with is similar to a simple if statement:

{{ with PIPELINE }}

# restricted scope

{{ end }}

Scopes can be changed. with can allow you to set the current scope (.) to a particular object. For example, we’ve been working with .Values.favorites. Let’s rewrite our ConfigMap to alter the . scope to point to .Values.favorites:

apiVersion: v1

kind: ConfigMap

metadata:

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

data:

myvalue: "Hello World"

{{- with .Values.favorite }}

drink: {{ .drink | default "tea" | quote }}

food: {{ .food | upper | quote }}

{{- end }}

(Note that we removed the if conditional from the previous exercise)

Notice that now we can reference .drink and .food without qualifying them. That is because the with statement sets . to point to .Values.favorite. The . is reset to its previous scope after {{ end }}.

But here’s a note of caution! Inside of the restricted scope, you will not be able to access the other objects from the parent scope. This, for example, will fail:

{{- with .Values.favorite }}

drink: {{ .drink | default "tea" | quote }}

food: {{ .food | upper | quote }}

release: {{ .Release.Name }}

{{- end }}

It will produce an error because Release.Name is not inside of the restricted scope for .. However, if we swap the last two lines, all will work as expected because the scope is reset after {{end}}.

{{- with .Values.favorite }}

drink: {{ .drink | default "tea" | quote }}

food: {{ .food | upper | quote }}

{{- end }}

release: {{ .Release.Name }}

After looking at range, we will take a look at template variables, which offers one solution to the scoping issue above.

**LOOPING WITH THE**RANGE**ACTION**

Many programming languages have support for looping using for loops, foreach loops, or similar functional mechanisms. In Helm’s template language, the way to iterate through a collection is to use the range operator.

To start, let’s add a list of pizza toppings to our values.yaml file:

favorite:

drink: coffee

food: pizza

pizzaToppings:

- mushrooms

- cheese

- peppers

- onions

Now we have a list (called a slice in templates) of pizzaToppings. We can modify our template to print this list into our ConfigMap:

apiVersion: v1

kind: ConfigMap

metadata:

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

data:

myvalue: "Hello World"

{{- with .Values.favorite }}

drink: {{ .drink | default "tea" | quote }}

food: {{ .food | upper | quote }}

{{- end }}

toppings: |-

{{- range .Values.pizzaToppings }}

- {{ . | title | quote }}

{{- end }}

Let’s take a closer look at the toppings: list. The range function will “range over” (iterate through) the pizzaToppings list. But now something interesting happens. Just like with sets the scope of ., so does a range operator. Each time through the loop, . is set to the current pizza topping. That is, the first time, . is set to mushrooms. The second iteration it is set to cheese, and so on.

We can send the value of . directly down a pipeline, so when we do {{ . | title | quote }}, it sends . to title (title case function) and then to quote. If we run this template, the output will be:

# Source: mychart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: edgy-dragonfly-configmap

data:

myvalue: "Hello World"

drink: "coffee"

food: "PIZZA"

toppings: |-

- "Mushrooms"

- "Cheese"

- "Peppers"

- "Onions"

Now, in this example we’ve done something tricky. The toppings: |- line is declaring a multi-line string. So our list of toppings is actually not a YAML list. It’s a big string. Why would we do this? Because the data in ConfigMaps data is composed of key/value pairs, where both the key and the value are simple strings. To understand why this is the case, take a look at the [Kubernetes ConfigMap docs](https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/). For us, though, this detail doesn’t matter much.

The |- marker in YAML takes a multi-line string. This can be a useful technique for embedding big blocks of data inside of your manifests, as exemplified here.

Sometimes it’s useful to be able to quickly make a list inside of your template, and then iterate over that list. Helm templates have a function that’s called just that: list.

sizes: |-

{{- range list "small" "medium" "large" }}

- {{ . }}

{{- end }}

The above will produce this:

sizes: |-

- small

- medium

- large

In addition to lists, range can be used to iterate over collections that have a key and a value (like a map or dict). We’ll see how to do that in the next section when we introduce template variables.

**Variables**

With functions, pipelines, objects, and control structures under our belts, we can turn to one of the more basic ideas in many programming languages: variables. In templates, they are less frequently used. But we will see how to use them to simplify code, and to make better use of with and range.

In an earlier example, we saw that this code will fail:

{{- with .Values.favorite }}

drink: {{ .drink | default "tea" | quote }}

food: {{ .food | upper | quote }}

release: {{ .Release.Name }}

{{- end }}

Release.Name is not inside of the scope that’s restricted in the with block. One way to work around scoping issues is to assign objects to variables that can be accessed without respect to the present scope.

In Helm templates, a variable is a named reference to another object. It follows the form $name. Variables are assigned with a special assignment operator: :=. We can rewrite the above to use a variable for Release.Name.

apiVersion: v1

kind: ConfigMap

metadata:

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

data:

myvalue: "Hello World"

{{- $relname := .Release.Name -}}

{{- with .Values.favorite }}

drink: {{ .drink | default "tea" | quote }}

food: {{ .food | upper | quote }}

release: {{ $relname }}

{{- end }}

Notice that before we start the with block, we assign $relname := .Release.Name. Now inside of the with block, the $relname variable still points to the release name.

Running that will produce this:

# Source: mychart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: viable-badger-configmap

data:

myvalue: "Hello World"

drink: "coffee"

food: "PIZZA"

release: viable-badger

Variables are particularly useful in range loops. They can be used on list-like objects to capture both the index and the value:

toppings: |-

{{- range $index, $topping := .Values.pizzaToppings }}

{{ $index }}: {{ $topping }}

{{- end }}

Note that range comes first, then the variables, then the assignment operator, then the list. This will assign the integer index (starting from zero) to $index and the value to $topping. Running it will produce:

toppings: |-

0: mushrooms

1: cheese

2: peppers

3: onions

For data structures that have both a key and a value, we can use range to get both. For example, we can loop through .Values.favorite like this:

apiVersion: v1

kind: ConfigMap

metadata:

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

data:

myvalue: "Hello World"

{{- range $key, $val := .Values.favorite }}

{{ $key }}: {{ $val | quote }}

{{- end}}

Now on the first iteration, $key will be drink and $val will be coffee, and on the second, $key will be food and $val will be pizza. Running the above will generate this:

# Source: mychart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: eager-rabbit-configmap

data:

myvalue: "Hello World"

drink: "coffee"

food: "pizza"

Variables are normally not “global”. They are scoped to the block in which they are declared. Earlier, we assigned $relname in the top level of the template. That variable will be in scope for the entire template. But in our last example, $key and $val will only be in scope inside of the {{range...}}{{end}} block.

However, there is one variable that is always global - $ - this variable will always point to the root context. This can be very useful when you are looping in a range and need to know the chart’s release name.

An example illustrating this:

{{- range .Values.tlsSecrets }}

apiVersion: v1

kind: Secret

metadata:

name: {{ .name }}

labels:

# Many helm templates would use `.` below, but that will not work,

# however `$` will work here

app.kubernetes.io/name: {{ template "fullname" $ }}

# I cannot reference .Chart.Name, but I can do $.Chart.Name

helm.sh/chart: "{{ $.Chart.Name }}-{{ $.Chart.Version }}"

app.kubernetes.io/instance: "{{ $.Release.Name }}"

# Value from appVersion in Chart.yaml

app.kubernetes.io/version: "{{ $.Chart.AppVersion }}"

app.kubernetes.io/managed-by: "{{ $.Release.Service }}"

type: kubernetes.io/tls

data:

tls.crt: {{ .certificate }}

tls.key: {{ .key }}

---

{{- end }}

So far we have looked at just one template declared in just one file. But one of the powerful features of the Helm template language is its ability to declare multiple templates and use them together. We’ll turn to that in the next section.

**Named Templates**

It is time to move beyond one template, and begin to create others. In this section, we will see how to define *named templates* in one file, and then use them elsewhere. A *named template* (sometimes called a *partial* or a *subtemplate*) is simply a template defined inside of a file, and given a name. We’ll see two ways to create them, and a few different ways to use them.

In the “Flow Control” section we introduced three actions for declaring and managing templates: define, template, and block. In this section, we’ll cover those three actions, and also introduce a special-purpose include function that works similarly to the template action.

An important detail to keep in mind when naming templates: **template names are global**. If you declare two templates with the same name, whichever one is loaded last will be the one used. Because templates in subcharts are compiled together with top-level templates, you should be careful to name your templates with *chart-specific names*.

One popular naming convention is to prefix each defined template with the name of the chart: {{ define "mychart.labels" }}. By using the specific chart name as a prefix we can avoid any conflicts that may arise due to two different charts that implement templates of the same name.

**PARTIALS AND**\_**FILES**

So far, we’ve used one file, and that one file has contained a single template. But Helm’s template language allows you to create named embedded templates, that can be accessed by name elsewhere.

Before we get to the nuts-and-bolts of writing those templates, there is file naming convention that deserves mention:

* Most files in templates/ are treated as if they contain Kubernetes manifests
* The NOTES.txt is one exception
* But files whose name begins with an underscore (\_) are assumed to *not* have a manifest inside. These files are not rendered to Kubernetes object definitions, but are available everywhere within other chart templates for use.

These files are used to store partials and helpers. In fact, when we first created mychart, we saw a file called \_helpers.tpl. That file is the default location for template partials.

**DECLARING AND USING TEMPLATES WITH**DEFINE**AND**TEMPLATE

The define action allows us to create a named template inside of a template file. Its syntax goes like this:

{{ define "MY.NAME" }}

# body of template here

{{ end }}

For example, we can define a template to encapsulate a Kubernetes block of labels:

{{- define "mychart.labels" }}

labels:

generator: helm

date: {{ now | htmlDate }}

{{- end }}

Now we can embed this template inside of our existing ConfigMap, and then include it with the template action:

{{- define "mychart.labels" }}

labels:

generator: helm

date: {{ now | htmlDate }}

{{- end }}

apiVersion: v1

kind: ConfigMap

metadata:

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

{{- template "mychart.labels" }}

data:

myvalue: "Hello World"

{{- range $key, $val := .Values.favorite }}

{{ $key }}: {{ $val | quote }}

{{- end }}

When the template engine reads this file, it will store away the reference to mychart.labels until template "mychart.labels" is called. Then it will render that template inline. So the result will look like this:

# Source: mychart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: running-panda-configmap

labels:

generator: helm

date: 2016-11-02

data:

myvalue: "Hello World"

drink: "coffee"

food: "pizza"

Conventionally, Helm charts put these templates inside of a partials file, usually \_helpers.tpl. Let’s move this function there:

{{/\* Generate basic labels \*/}}

{{- define "mychart.labels" }}

labels:

generator: helm

date: {{ now | htmlDate }}

{{- end }}

By convention, define functions should have a simple documentation block ({{/\* ... \*/}}) describing what they do.

Even though this definition is in \_helpers.tpl, it can still be accessed in configmap.yaml:

apiVersion: v1

kind: ConfigMap

metadata:

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

{{- template "mychart.labels" }}

data:

myvalue: "Hello World"

{{- range $key, $val := .Values.favorite }}

{{ $key }}: {{ $val | quote }}

{{- end }}

As mentioned above, **template names are global**. As a result of this, if two templates are declared with the same name the last occurrence will be the one that is used. Since templates in subcharts are compiled together with top-level templates, it is best to name your templates with *chart specific names*. A popular naming convention is to prefix each defined template with the name of the chart: {{ define "mychart.labels" }}.

**SETTING THE SCOPE OF A TEMPLATE**

In the template we defined above, we did not use any objects. We just used functions. Let’s modify our defined template to include the chart name and chart version:

{{/\* Generate basic labels \*/}}

{{- define "mychart.labels" }}

labels:

generator: helm

date: {{ now | htmlDate }}

chart: {{ .Chart.Name }}

version: {{ .Chart.Version }}

{{- end }}

If we render this, the result will not be what we expect:

# Source: mychart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: moldy-jaguar-configmap

labels:

generator: helm

date: 2016-11-02

chart:

version:

What happened to the name and version? They weren’t in the scope for our defined template. When a named template (created with define) is rendered, it will receive the scope passed in by the template call. In our example, we included the template like this:

{{- template "mychart.labels" }}

No scope was passed in, so within the template we cannot access anything in .. This is easy enough to fix, though. We simply pass a scope to the template:

apiVersion: v1

kind: ConfigMap

metadata:

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

{{- template "mychart.labels" . }}

Note that we pass . at the end of the template call. We could just as easily pass .Values or .Values.favorite or whatever scope we want. But what we want is the top-level scope.

Now when we execute this template with helm install --dry-run --debug ./mychart, we get this:

# Source: mychart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: plinking-anaco-configmap

labels:

generator: helm

date: 2016-11-02

chart: mychart

version: 0.1.0

Now {{ .Chart.Name }} resolves to mychart, and {{ .Chart.Version }} resolves to 0.1.0.

**THE**INCLUDE**FUNCTION**

Say we’ve defined a simple template that looks like this:

{{- define "mychart.app" -}}

app\_name: {{ .Chart.Name }}

app\_version: "{{ .Chart.Version }}+{{ .Release.Time.Seconds }}"

{{- end -}}

Now say I want to insert this both into the labels: section of my template, and also the data: section:

apiVersion: v1

kind: ConfigMap

metadata:

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

labels:

{{ template "mychart.app" .}}

data:

myvalue: "Hello World"

{{- range $key, $val := .Values.favorite }}

{{ $key }}: {{ $val | quote }}

{{- end }}

{{ template "mychart.app" . }}

The output will not be what we expect:

# Source: mychart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: measly-whippet-configmap

labels:

app\_name: mychart

app\_version: "0.1.0+1478129847"

data:

myvalue: "Hello World"

drink: "coffee"

food: "pizza"

app\_name: mychart

app\_version: "0.1.0+1478129847"

Note that the indentation on app\_version is wrong in both places. Why? Because the template that is substituted in has the text aligned to the right. Because template is an action, and not a function, there is no way to pass the output of a template call to other functions; the data is simply inserted inline.

To work around this case, Helm provides an alternative to template that will import the contents of a template into the present pipeline where it can be passed along to other functions in the pipeline.

Here’s the example above, corrected to use nindent to indent the mychart\_app template correctly:

apiVersion: v1

kind: ConfigMap

metadata:

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

labels:

{{- include "mychart.app" . | nindent 4 }}

data:

myvalue: "Hello World"

{{- range $key, $val := .Values.favorite }}

{{ $key }}: {{ $val | quote }}

{{- end }}

{{- include "mychart.app" . | nindent 2 }}

Now the produced YAML is correctly indented for each section:

# Source: mychart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: edgy-mole-configmap

labels:

app\_name: mychart

app\_version: "0.1.0+1478129987"

data:

myvalue: "Hello World"

drink: "coffee"

food: "pizza"

app\_name: mychart

app\_version: "0.1.0+1478129987"

It is considered preferable to use include over template in Helm templates simply so that the output formatting can be handled better for YAML documents.

Sometimes we want to import content, but not as templates. That is, we want to import files verbatim. We can achieve this by accessing files through the .Files object described in the next section.

**Accessing Files Inside Templates**

In the previous section we looked at several ways to create and access named templates. This makes it easy to import one template from within another template. But sometimes it is desirable to import a *file that is not a template* and inject its contents without sending the contents through the template renderer.

Helm provides access to files through the .Files object. Before we get going with the template examples, though, there are a few things to note about how this works:

* It is okay to add extra files to your Helm chart. These files will be bundled and sent to Tiller. Be careful, though. Charts must be smaller than 1M because of the storage limitations of Kubernetes objects.
* Some files cannot be accessed through the .Files object, usually for security reasons.
  + Files in templates/ cannot be accessed.
  + Files excluded using .helmignore cannot be accessed.
* Charts do not preserve UNIX mode information, so file-level permissions will have no impact on the availability of a file when it comes to the .Files object.
* [Basic example](https://helm.sh/docs/chart_template_guide/#basic-example)
* [Path helpers](https://helm.sh/docs/chart_template_guide/#path-helpers)
* [Glob patterns](https://helm.sh/docs/chart_template_guide/#glob-patterns)
* [ConfigMap and Secrets utility functions](https://helm.sh/docs/chart_template_guide/#configmap-and-secrets-utility-functions)
* [Encoding](https://helm.sh/docs/chart_template_guide/#encoding)
* [Lines](https://helm.sh/docs/chart_template_guide/#lines)

**BASIC EXAMPLE**

With those caveats behind, let’s write a template that reads three files into our ConfigMap. To get started, we will add three files to the chart, putting all three directly inside of the mychart/ directory.

config1.toml:

message = Hello from config 1

config2.toml:

message = This is config 2

config3.toml:

message = Goodbye from config 3

Each of these is a simple TOML file (think old-school Windows INI files). We know the names of these files, so we can use a range function to loop through them and inject their contents into our ConfigMap.

apiVersion: v1

kind: ConfigMap

metadata:

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

data:

{{- $files := .Files }}

{{- range list "config1.toml" "config2.toml" "config3.toml" }}

{{ . }}: |-

{{ $files.Get . }}

{{- end }}

This config map uses several of the techniques discussed in previous sections. For example, we create a $files variable to hold a reference to the .Files object. We also use the list function to create a list of files that we loop through. Then we print each file name ({{.}}: |-) followed by the contents of the file {{ $files.Get . }}.

Running this template will produce a single ConfigMap with the contents of all three files:

# Source: mychart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: quieting-giraf-configmap

data:

config1.toml: |-

message = Hello from config 1

config2.toml: |-

message = This is config 2

config3.toml: |-

message = Goodbye from config 3

**PATH HELPERS**

When working with files, it can be very useful to perform some standard operations on the file paths themselves. To help with this, Helm imports many of the functions from Go’s [path](https://golang.org/pkg/path/) package for your use. They are all accessible with the same names as in the Go package, but with a lowercase first letter. For example, Base becomes base, etc.

The imported functions are:

* Base
* Dir
* Ext
* IsAbs
* Clean

**GLOB PATTERNS**

As your chart grows, you may find you have a greater need to organize your files more, and so we provide a Files.Glob(pattern string) method to assist in extracting certain files with all the flexibility of [glob patterns](https://godoc.org/github.com/gobwas/glob).

.Glob returns a Files type, so you may call any of the Files methods on the returned object.

For example, imagine the directory structure:

foo/:

foo.txt foo.yaml

bar/:

bar.go bar.conf baz.yaml

You have multiple options with Globs:

{{ $root := . }}

{{ range $path, $bytes := .Files.Glob "\*\*.yaml" }}

{{ $path }}: |-

{{ $root.Files.Get $path }}

{{ end }}

Or

{{ range $path, $bytes := .Files.Glob "foo/\*" }}

{{ base $path }}: '{{ $root.Files.Get $path | b64enc }}'

{{ end }}

**CONFIGMAP AND SECRETS UTILITY FUNCTIONS**

(Not present in version 2.0.2 or prior)

It is very common to want to place file content into both configmaps and secrets, for mounting into your pods at run time. To help with this, we provide a couple utility methods on the Files type.

For further organization, it is especially useful to use these methods in conjunction with the Glob method.

Given the directory structure from the [Glob](https://helm.sh/docs/chart_template_guide/#glob-patterns) example above:

apiVersion: v1

kind: ConfigMap

metadata:

name: conf

data:

{{- (.Files.Glob "foo/\*").AsConfig | nindent 2 }}

---

apiVersion: v1

kind: Secret

metadata:

name: very-secret

type: Opaque

data:

{{- (.Files.Glob "bar/\*").AsSecrets | nindent 2 }}

**ENCODING**

You can import a file and have the template base-64 encode it to ensure successful transmission:

apiVersion: v1

kind: Secret

metadata:

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

type: Opaque

data:

token: |-

{{ .Files.Get "config1.toml" | b64enc }}

The above will take the same config1.toml file we used before and encode it:

# Source: mychart/templates/secret.yaml

apiVersion: v1

kind: Secret

metadata:

name: lucky-turkey-secret

type: Opaque

data:

token: |-

bWVzc2FnZSA9IEhlbGxvIGZyb20gY29uZmlnIDEK

**LINES**

Sometimes it is desirable to access each line of a file in your template. We provide a convenient Lines method for this.

data:

some-file.txt: {{ range .Files.Lines "foo/bar.txt" }}

{{ . }}{{ end }}

Currently, there is no way to pass files external to the chart during helm install. So if you are asking users to supply data, it must be loaded using helm install -f or helm install --set.

This discussion wraps up our dive into the tools and techniques for writing Helm templates. In the next section we will see how you can use one special file, templates/NOTES.txt, to send post-installation instructions to the users of your chart.

**Creating a NOTES.txt File**

In this section we are going to look at Helm’s tool for providing instructions to your chart users. At the end of a helm install or helm upgrade, Helm can print out a block of helpful information for users. This information is highly customizable using templates.

To add installation notes to your chart, simply create a templates/NOTES.txt file. This file is plain text, but it is processed like as a template, and has all the normal template functions and objects available.

Let’s create a simple NOTES.txt file:

Thank you for installing {{ .Chart.Name }}.

Your release is named {{ .Release.Name }}.

To learn more about the release, try:

$ helm status {{ .Release.Name }}

$ helm get {{ .Release.Name }}

Now if we run helm install ./mychart we will see this message at the bottom:

RESOURCES:

==> v1/Secret

NAME TYPE DATA AGE

rude-cardinal-secret Opaque 1 0s

==> v1/ConfigMap

NAME DATA AGE

rude-cardinal-configmap 3 0s

NOTES:

Thank you for installing mychart.

Your release is named rude-cardinal.

To learn more about the release, try:

$ helm status rude-cardinal

$ helm get rude-cardinal

Using NOTES.txt this way is a great way to give your users detailed information about how to use their newly installed chart. Creating a NOTES.txt file is strongly recommended, though it is not required.

**Subcharts and Global Values**

To this point we have been working only with one chart. But charts can have dependencies, called *subcharts*, that also have their own values and templates. In this section we will create a subchart and see the different ways we can access values from within templates.

Before we dive into the code, there are a few important details to learn about subcharts.

1. A subchart is considered “stand-alone”, which means a subchart can never explicitly depend on its parent chart.
2. For that reason, a subchart cannot access the values of its parent.
3. A parent chart can override values for subcharts.
4. Helm has a concept of *global values* that can be accessed by all charts.

As we walk through the examples in this section, many of these concepts will become clearer.

**CREATING A SUBCHART**

For these exercises, we’ll start with the mychart/ chart we created at the beginning of this guide, and we’ll add a new chart inside of it.

$ cd mychart/charts

$ helm create mysubchart

Creating mysubchart

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

Notice that just as before, we deleted all of the base templates so that we can start from scratch. In this guide, we are focused on how templates work, not on managing dependencies. But the [Charts Guide](https://helm.sh/docs/chart_template_guide/#../charts) has more information on how subcharts work.

**ADDING VALUES AND A TEMPLATE TO THE SUBCHART**

Next, let’s create a simple template and values file for our mysubchart chart. There should already be a values.yaml in mychart/charts/mysubchart. We’ll set it up like this:

dessert: cake

Next, we’ll create a new ConfigMap template in mychart/charts/mysubchart/templates/configmap.yaml:

apiVersion: v1

kind: ConfigMap

metadata:

name: {{ .Release.Name }}-cfgmap2

data:

dessert: {{ .Values.dessert }}

Because every subchart is a *stand-alone chart*, we can test mysubchart on its own:

$ helm install --dry-run --debug mychart/charts/mysubchart

SERVER: "localhost:44134"

CHART PATH: /Users/mattbutcher/Code/Go/src/k8s.io/helm/\_scratch/mychart/charts/mysubchart

NAME: newbie-elk

TARGET NAMESPACE: default

CHART: mysubchart 0.1.0

MANIFEST:

---

# Source: mysubchart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: newbie-elk-cfgmap2

data:

dessert: cake

**OVERRIDING VALUES OF A CHILD CHART**

Our original chart, mychart is now the *parent* chart of mysubchart. This relationship is based entirely on the fact that mysubchart is within mychart/charts.

Because mychart is a parent, we can specify configuration in mychart and have that configuration pushed into mysubchart. For example, we can modify mychart/values.yaml like this:

favorite:

drink: coffee

food: pizza

pizzaToppings:

- mushrooms

- cheese

- peppers

- onions

mysubchart:

dessert: ice cream

Note the last two lines. Any directives inside of the mysubchart section will be sent to the mysubchart chart. So if we run helm install --dry-run --debug mychart, one of the things we will see is the mysubchart ConfigMap:

# Source: mychart/charts/mysubchart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: unhinged-bee-cfgmap2

data:

dessert: ice cream

The value at the top level has now overridden the value of the subchart.

There’s an important detail to notice here. We didn’t change the template of mychart/charts/mysubchart/templates/configmap.yaml to point to .Values.mysubchart.dessert. From that template’s perspective, the value is still located at .Values.dessert. As the template engine passes values along, it sets the scope. So for the mysubchart templates, only values specifically for mysubchart will be available in .Values.

Sometimes, though, you do want certain values to be available to all of the templates. This is accomplished using global chart values.

**GLOBAL CHART VALUES**

Global values are values that can be accessed from any chart or subchart by exactly the same name. Globals require explicit declaration. You can’t use an existing non-global as if it were a global.

The Values data type has a reserved section called Values.global where global values can be set. Let’s set one in our mychart/values.yaml file.

favorite:

drink: coffee

food: pizza

pizzaToppings:

- mushrooms

- cheese

- peppers

- onions

mysubchart:

dessert: ice cream

global:

salad: caesar

Because of the way globals work, both mychart/templates/configmap.yaml and mychart/charts/mysubchart/templates/configmap.yaml should be able to access that value as {{ .Values.global.salad}}.

mychart/templates/configmap.yaml:

apiVersion: v1

kind: ConfigMap

metadata:

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

data:

salad: {{ .Values.global.salad }}

mychart/charts/mysubchart/templates/configmap.yaml:

apiVersion: v1

kind: ConfigMap

metadata:

name: {{ .Release.Name }}-cfgmap2

data:

dessert: {{ .Values.dessert }}

salad: {{ .Values.global.salad }}

Now if we run a dry run install, we’ll see the same value in both outputs:

# Source: mychart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: silly-snake-configmap

data:

salad: caesar

---

# Source: mychart/charts/mysubchart/templates/configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: silly-snake-cfgmap2

data:

dessert: ice cream

salad: caesar

Globals are useful for passing information like this, though it does take some planning to make sure the right templates are configured to use globals.

**SHARING TEMPLATES WITH SUBCHARTS**

Parent charts and subcharts can share templates. Any defined block in any chart is available to other charts.

For example, we can define a simple template like this:

{{- define "labels" }}from: mychart{{ end }}

Recall how the labels on templates are *globally shared*. Thus, the labels chart can be included from any other chart.

While chart developers have a choice between include and template, one advantage of using include is that include can dynamically reference templates:

{{ include $mytemplate }}

The above will dereference $mytemplate. The template function, in contrast, will only accept a string literal.

**AVOID USING BLOCKS**

The Go template language provides a block keyword that allows developers to provide a default implementation which is overridden later. In Helm charts, blocks are not the best tool for overriding because if multiple implementations of the same block are provided, the one selected is unpredictable.

The suggestion is to instead use include.

**Debugging Templates**

Debugging templates can be tricky simply because the templates are rendered on the Tiller server, not the Helm client. And then the rendered templates are sent to the Kubernetes API server, which may reject the YAML files for reasons other than formatting.

There are a few commands that can help you debug.

* helm lint is your go-to tool for verifying that your chart follows best practices
* helm install --dry-run --debug: We’ve seen this trick already. It’s a great way to have the server render your templates, then return the resulting manifest file.
* helm get manifest: This is a good way to see what templates are installed on the server.

When your YAML is failing to parse, but you want to see what is generated, one easy way to retrieve the YAML is to comment out the problem section in the template, and then re-run helm install --dry-run --debug:

apiVersion: v1

# some: problem section

# {{ .Values.foo | quote }}

The above will be rendered and returned with the comments intact:

apiVersion: v1

# some: problem section

# "bar"

This provides a quick way of viewing the generated content without YAML parse errors blocking.

**Wrapping Up**

This guide is intended to give you, the chart developer, a strong understanding of how to use Helm’s template language. The guide focuses on the technical aspects of template development.

But there are many things this guide has not covered when it comes to the practical day-to-day development of charts. Here are some useful pointers to other documentation that will help you as you create new charts:

* The [Helm Charts project](https://github.com/helm/charts) is an indispensable source of charts. That project is also sets the standard for best practices in chart development.
* The Kubernetes [Documentation](https://kubernetes.io/docs/home/) provides detailed examples of the various resource kinds that you can use, from ConfigMaps and Secrets to DaemonSets and Deployments.
* The Helm [Charts Guide](https://helm.sh/docs/chart_template_guide/#../charts) explains the workflow of using charts.
* The Helm [Chart Hooks Guide](https://helm.sh/docs/chart_template_guide/#../charts_hooks) explains how to create lifecycle hooks.
* The Helm [Charts Tips and Tricks](https://helm.sh/docs/chart_template_guide/#../chart-development-tips-and-tricks) article provides some useful tips for writing charts.
* The [Sprig documentation](https://github.com/Masterminds/sprig) documents more than sixty of the template functions.
* The [Go template docs](https://godoc.org/text/template) explain the template syntax in detail.
* The [Schelm tool](https://github.com/databus23/schelm) is a nice helper utility for debugging charts.

Sometimes it’s easier to ask a few questions and get answers from experienced developers. The best place to do this is in the [Kubernetes Slack](https://kubernetes.slack.com/) Helm channels:

* [#helm-users](https://kubernetes.slack.com/messages/helm-users)
* [#helm-dev](https://kubernetes.slack.com/messages/helm-dev)
* [#charts](https://kubernetes.slack.com/messages/charts)

Finally, if you find errors or omissions in this document, want to suggest some new content, or would like to contribute, visit [The Helm Project](https://github.com/helm/helm).

**YAML Techniques**

Most of this guide has been focused on writing the template language. Here, we’ll look at the YAML format. YAML has some useful features that we, as template authors, can use to make our templates less error prone and easier to read.

**SCALARS AND COLLECTIONS**

According to the [YAML spec](https://yaml.org/spec/1.2/spec.html), there are two types of collections, and many scalar types.

The two types of collections are maps and sequences:

map:

one: 1

two: 2

three: 3

sequence:

- one

- two

- three

Scalar values are individual values (as opposed to collections)

**Scalar Types in YAML**

In Helm’s dialect of YAML, the scalar data type of a value is determined by a complex set of rules, including the Kubernetes schema for resource definitions. But when inferring types, the following rules tend to hold true.

If an integer or float is an unquoted bare word, it is typically treated as a numeric type:

count: 1

size: 2.34

But if they are quoted, they are treated as strings:

count: "1" # <-- string, not int

size: '2.34' # <-- string, not float

The same is true of booleans:

isGood: true # bool

answer: "true" # string

The word for an empty value is null (not nil).

Note that port: "80" is valid YAML, and will pass through both the template engine and the YAML parser, but will fail if Kubernetes expects port to be an integer.

In some cases, you can force a particular type inference using YAML node tags:

coffee: "yes, please"

age: !!str 21

port: !!int "80"

In the above, !!str tells the parser that age is a string, even if it looks like an int. And port is treated as an int, even though it is quoted.

**STRINGS IN YAML**

Much of the data that we place in YAML documents are strings. YAML has more than one way to represent a string. This section explains the ways and demonstrates how to use some of them.

There are three “inline” ways of declaring a string:

way1: bare words

way2: "double-quoted strings"

way3: 'single-quoted strings'

All inline styles must be on one line.

* Bare words are unquoted, and are not escaped. For this reason, you have to be careful what characters you use.
* Double-quoted strings can have specific characters escaped with \. For example "\"Hello\", she said". You can escape line breaks with \n.
* Single-quoted strings are “literal” strings, and do not use the \ to escape characters. The only escape sequence is '', which is decoded as a single '.

In addition to the one-line strings, you can declare multi-line strings:

coffee: |

Latte

Cappuccino

Espresso

The above will treat the value of coffee as a single string equivalent to Latte\nCappuccino\nEspresso\n.

Note that the first line after the | must be correctly indented. So we could break the example above by doing this:

coffee: |

Latte

Cappuccino

Espresso

Because Latte is incorrectly indented, we’d get an error like this:

Error parsing file: error converting YAML to JSON: yaml: line 7: did not find expected key

In templates, it is sometimes safer to put a fake “first line” of content in a multi-line document just for protection from the above error:

coffee: |

# Commented first line

Latte

Cappuccino

Espresso

Note that whatever that first line is, it will be preserved in the output of the string. So if you are, for example, using this technique to inject a file’s contents into a ConfigMap, the comment should be of the type expected by whatever is reading that entry.

**Controlling Spaces in Multi-line Strings**

In the example above, we used | to indicate a multi-line string. But notice that the content of our string was followed with a trailing \n. If we want the YAML processor to strip off the trailing newline, we can add a - after the |:

coffee: |-

Latte

Cappuccino

Espresso

Now the coffee value will be: Latte\nCappuccino\nEspresso (with no trailing \n).

Other times, we might want all trailing whitespace to be preserved. We can do this with the |+ notation:

coffee: |+

Latte

Cappuccino

Espresso

another: value

Now the value of coffee will be Latte\nCappuccino\nEspresso\n\n\n.

Indentation inside of a text block is preserved, and results in the preservation of line breaks, too:

coffee: |-

Latte

12 oz

16 oz

Cappuccino

Espresso

In the above case, coffee will be Latte\n 12 oz\n 16 oz\nCappuccino\nEspresso.

**Indenting and Templates**

When writing templates, you may find yourself wanting to inject the contents of a file into the template. As we saw in previous chapters, there are two ways of doing this:

* Use {{ .Files.Get "FILENAME" }} to get the contents of a file in the chart.
* Use {{ include "TEMPLATE" . }} to render a template and then place its contents into the chart.

When inserting files into YAML, it’s good to understand the multi-line rules above. Often times, the easiest way to insert a static file is to do something like this:

myfile: |

{{ .Files.Get "myfile.txt" | indent 2 }}

Note how we do the indentation above: indent 2 tells the template engine to indent every line in “myfile.txt” with two spaces. Note that we do not indent that template line. That’s because if we did, the file content of the first line would be indented twice.

**Folded Multi-line Strings**

Sometimes you want to represent a string in your YAML with multiple lines, but want it to be treated as one long line when it is interpreted. This is called “folding”. To declare a folded block, use > instead of |:

coffee: >

Latte

Cappuccino

Espresso

The value of coffee above will be Latte Cappuccino Espresso\n. Note that all but the last line feed will be converted to spaces. You can combine the whitespace controls with the folded text marker, so >- will replace or trim all newlines.

Note that in the folded syntax, indenting text will cause lines to be preserved.

coffee: >-

Latte

12 oz

16 oz

Cappuccino

Espresso

The above will produce Latte\n 12 oz\n 16 oz\nCappuccino Espresso. Note that both the spacing and the newlines are still there.

**EMBEDDING MULTIPLE DOCUMENTS IN ONE FILE**

It is possible to place more than one YAML documents into a single file. This is done by prefixing a new document with --- and ending the document with ...

---

document:1

...

---

document: 2

...

In many cases, either the --- or the ... may be omitted.

Some files in Helm cannot contain more than one doc. If, for example, more than one document is provided inside of a values.yaml file, only the first will be used.

Template files, however, may have more than one document. When this happens, the file (and all of its documents) is treated as one object during template rendering. But then the resulting YAML is split into multiple documents before it is fed to Kubernetes.

We recommend only using multiple documents per file when it is absolutely necessary. Having multiple documents in a file can be difficult to debug.

**YAML IS A SUPERSET OF JSON**

Because YAML is a superset of JSON, any valid JSON document *should* be valid YAML.

{

"coffee": "yes, please",

"coffees": [

"Latte", "Cappuccino", "Espresso"

]

}

The above is another way of representing this:

coffee: yes, please

coffees:

- Latte

- Cappuccino

- Espresso

And the two can be mixed (with care):

coffee: "yes, please"

coffees: [ "Latte", "Cappuccino", "Espresso"]

All three of these should parse into the same internal representation.

While this means that files such as values.yaml may contain JSON data, Helm does not treat the file extension .json as a valid suffix.

**YAML ANCHORS**

The YAML spec provides a way to store a reference to a value, and later refer to that value by reference. YAML refers to this as “anchoring”:

coffee: "yes, please"

favorite: &favoriteCoffee "Cappucino"

coffees:

- Latte

- \*favoriteCoffee

- Espresso

In the above, &favoriteCoffee sets a reference to Cappuccino. Later, that reference is used as \*favoriteCoffee. So coffees becomes Latte, Cappuccino, Espresso.

While there are a few cases where anchors are useful, there is one aspect of them that can cause subtle bugs: The first time the YAML is consumed, the reference is expanded and then discarded.

So if we were to decode and then re-encode the example above, the resulting YAML would be:

coffee: yes, please

favorite: Cappucino

coffees:

- Latte

- Cappucino

- Espresso

Because Helm and Kubernetes often read, modify, and then rewrite YAML files, the anchors will be lost.

**Appendix: Go Data Types and Templates**

The Helm template language is implemented in the strongly typed Go programming language. For that reason, variables in templates are *typed*. For the most part, variables will be exposed as one of the following types:

* string: A string of text
* bool: a true or false
* int: An integer value (there are also 8, 16, 32, and 64 bit signed and unsigned variants of this)
* float64: a 64-bit floating point value (there are also 8, 16, and 32 bit varieties of this)
* a byte slice ([]byte), often used to hold (potentially) binary data
* struct: an object with properties and methods
* a slice (indexed list) of one of the previous types
* a string-keyed map (map[string]interface{}) where the value is one of the previous types

There are many other types in Go, and sometimes you will have to convert between them in your templates. The easiest way to debug an object’s type is to pass it through printf "%t" in a template, which will print the type. Also see the typeOf and kindOf functions.