Published in Level Up Coding

Md Shamim Follow
Oct 17, 2022 · 4 min read · Listen

Save

Helm — Flow Control

Helm “flow control” using if/else, with and range

Photo by Joseph Barrientos on Unsplash

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 this article, we will discuss **helm flow control** along with various examples.

If/else

We can control the flow of the helm chart using the **If/else** statement just like any other programming language.

```
{{ if PIPELINE }}
  # Do something
{{ else if OTHER PIPELINE }}
  # Do something else
{{ else }}
  # Default case
{{ end }}
```

A pipeline will be 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`)

Now, Let’s see how we can use **if/else** in a helm template.

Suppose, we have **values.yaml** file with the following entries :

```yaml
# values.yaml
configMap:
  data:
    mode: dark
    env: test

```

values.yaml hosted with ❤ by **GitHub** **view raw**

And now we will create a **configmap.yaml** template file, where we will use **if/else** for decision-making purposes:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  {{ if .Values.configMap.data.darkMode }}          # boolean check
  mode: dark
  {{ else }}
  mode: light
  {{ end }}
  {{ if eq .Values.configMap.data.env "prod"  }}    # string check
  env: prod
  {{ else if eq .Values.configMap.data.env "dev" }}
  env: dev
  {{ else }}
  env: test
  {{ end }}

```

configmap.yaml hosted with ❤ by **GitHub** **view raw**

Generate the template using the **helm template** command :

```
>> helm template ~/webserver
---
# Source: webserver/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: release-name-configmap
data:

  mode: light


  env: test

```

Controlling Whitespace

In the above demonstration, we can see there are some **blank lines/spaces.** To remove any leading spaces we can use a dash `{{-` before the **if/else** statement.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  {{- if .Values.configMap.data.darkMode }}          # boolean check
  mode: dark
  {{- else }}
  mode: light
  {{- end }}
  {{- if eq .Values.configMap.data.env "prod"  }}    # string check
  env: prod
  {{- else if eq .Values.configMap.data.env "dev" }}
  env: dev
  {{- else }}
  env: test
  {{- end }}

```

configmap.yaml hosted with ❤ by **GitHub** **view raw**

```
>> helm template ~/webserver

---
# Source: webserver/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: release-name-configmap
data:
  mode: light
  env: test
```

In the previous demonstration, we used “ **`eq`** ” function along with the **`if`** statement. We can use other logical and flow control functions as per needs. There is a collection of logic and flow control functions available.

Let’s try to use **`and`** function along with the **`if`** statement; For that, we will add a few more entries to the **values.yaml** file :

```yaml
# values.yaml

configMap:
  data:
    darkMode: true
    os: mac
    env: test

```

values.yaml hosted with ❤ by **GitHub** **view raw**

What we want to achieve is, If **`darkMode: true`** and **`os: mac`** defined in **values.yaml** file, then our **configmap.yaml** template file will print **`mode: dark`** otherwise **`mode: light`** will be printed.

We have to modify the **configmap.yaml** template file accordingly to achieve the above-mentioned goal:

```yaml
# configmap.yaml

apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  {{- if and (.Values.configMap.data.darkMode) ( eq .Values.configMap.data.os "mac") }}
  mode: dark
  {{- else }}
  mode: light
  {{- end }}
  {{- if eq .Values.configMap.data.env "prod"  }}
  env: prod
  {{- else if eq .Values.configMap.data.env "dev" }}
  env: dev
  {{- else }}
  env: test
  {{- end }}

```

configmap.yaml hosted with ❤ by **GitHub** **view raw**

In the above demonstration, we have used **`and`** function to make a decision as per the requirements.

Now, generate the template to verify that the **configmap.yaml** file is functioning perfectly or not:

```
>> helm template ~/webserver/

---
# Source: webserver/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: release-name-configmap
data:
  mode: dark
  env: test
```

Modifying scope using “with”

The next control structure we will discuss is the `with` action. This controls variable scoping. Previously, we have seen that `.Values` tells the template to find the objects inside the `Values` object within the chart. Here, `.` means current scope. Scopes can be changed, `with` allows us to set the current scope to a particular object.

```
{{ with PIPELINE }}

  {{- toYaml . | nindent 2 }}

{{ end }}
```

Within the `with` scope, `.` does not refer to the root objects. Inside the `with` object, `.` is used to access the scopes of the current object.

Let’s see some examples for a better understanding.

Example 1:
Suppose we have **values.yaml** file with the following entries:

```yaml
# values.yaml

configMap:
  data:
    env: test
    platfrom:
     - java
     - python
     - golang

```

values.yaml hosted with ❤ by GitHub view raw

We will create a **configmap.yaml** template, where `with` will be used for variable scoping:

```yaml
#configmap.yaml

apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  env: {{ .Values.configMap.data.env }}
  {{- with .Values.configMap.data.platfrom }}
  platfrom: {{- toYaml . | nindent 2 | upper  }}
  {{- end }}

```

configmap.yaml hosted with ❤ by GitHub view raw

Now, generate the template :

```
>> helm template ~/webserver
---
# Source: webserver/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: release-name-configmap
data:
  env: test
  platfrom:
  - JAVA
```

```
    - PYTHON
    - GOLANG
```

Example 2:

Let’s add some additional entries to the **values.yaml** file:

```yaml
# values.yaml

configMap:
  data:
    env: test
    platfrom:
     - java
     - python
     - golang
    conf:
      os: linux
      database: mongo

```

values.yaml hosted with ❤ by **GitHub** **view raw**

And then we will modify our **configmap.yaml** template file. So that we can retrieve the additional data added to the **values.yaml** file:

```yaml
# configmap.yaml

apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  env: {{ .Values.configMap.data.env }}
  {{- with .Values.configMap.data.platfrom }}
  platfrom: {{- toYaml . | nindent 2 | upper  }}
  {{- end }}
  {{- with .Values.configMap.data.conf }}
  operating-system: {{ .os }}
  database-name: {{ .database }}
  {{- end }}

```

configmap.yaml hosted with ❤ by **GitHub** **view raw**

Finally, generate the template:

```
>> helm template ~/webserver

# Source: webserver/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: release-name-configmap
data:
  env: test
  platfrom:
  - JAVA
  - PYTHON
  - GOLANG
  operating-system: linux
  database-name: mongo
```

As we discussed earlier, within a `with` block `.` referred to a particular object. But there may be cases where we might have a requirement to access root objects or other objects, which are not a part of the current scope. For example, if we write something like this:

```
{{- with .Values.configMap.data.conf }}
  operating-system: {{ .os }}
  database-name: {{ .database }}
  k8s-namespace: {{ .Release.Namespace }}
{{- end }}
```

Then the above code will throw an error like this:

```
Error: template: webserver/templates/configmap.yaml:14:28: executing
"webserver/templates/configmap.yaml" at <.Release.Namespace>: nil
pointer evaluating interface {}.Namespace
```

Because, we are referring to an object, which resides outside of the current scope.

To solve this issue we can use the **$** sign in front of the **Release** object. Because root scope is also represented by the **$** sign.

```
{{- with .Values.configMap.data.conf }}
  operating-system: {{ .os }}
  database-name: {{ .database }}
```

```
    k8s-namespace: {{ $.Release.Namespace }}
  {{- end }}
```

Now, the above code will work fine, as we added a **$** sign in front of the **Release** object.

Range

`range` is as similar to **for/foreach** loops, like other programming languages. In Helm's template language, the way to iterate through a collection is to use the `range` operator.

Suppose, we have **values.yaml** file with the following entries :

```
# values.yaml
configMap:
  data:
    env: test
    platfrom:
     - java
     - python
     - golang

```

values.yaml hosted with ❤ by **GitHub** **view raw**

In the **values.yaml** file we have a list of “**platform**” defined, let's create a **configmap.yaml** template file to retrieve the list using `range` operator:

```yaml
# configmap.yaml

apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  env: {{ .Values.configMap.data.env }}
  platfrom: |
  {{- range .Values.configMap.data.platfrom }}
   - {{ . | title | quote }}
  {{- end }}

```

configmap.yaml hosted with ❤ by **GitHub** **view raw**

Finally, generate the template :

```
>> helm template ~/webserver

---
# Source: webserver/templates/test.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: release-name-configmap
data:
  env: test
  platfrom: |
   - "Java"
   - "Python"
   - "Golang"
```

> If you found this article helpful, please **don’t forget** to hit the **Clap** and **Follow** buttons to help me write more articles like this.
> Thank You 🖤

All Articles on Helm Chart -

Control structures (called "actions" in template parlance) provide you, the template author, with the ability to...

helm. sh

Helm Helm Chart Kubernetes K 8 S Kubernetes Cluster