Sommario

Cos'è Helm

Helm è un tool di supporto nella creazione e gestione delle risorse di Kubernetes, nonché di tutte le altre piattaforme che implementano le API server di Kubernetes.
N.B. Questa guida è basata su Helm v3.7.1.

I concetti principali di Helm sono:

- Un Chart, che rappresenta un insieme di risorse Kubernetes necessarie per definire e creare un'istanza di un'applicazione su un cluster.
- Una Release, che rappresenta un'istanza in esecuzione di un Chart.

L'eseguibile di Helm fornisce due distinti componenti:

1. Un command-line client tramite il quale è possibile:

    - Sviluppare chart.
    - Gestire repositories dove i chart possono essere presi e/o salvati.
    - Gestire le releases.
    - Interfacciarsi con le librerie di Helm.

2. Le librerie di Helm che, scritte in Go, interagiscono con le API server di Kubernetes per eseguire le seguenti operazioni su un cluster Kubernetes:

    - Installare un chart.
    - Aggiornare un chart.
    - Disinstallare un chart.

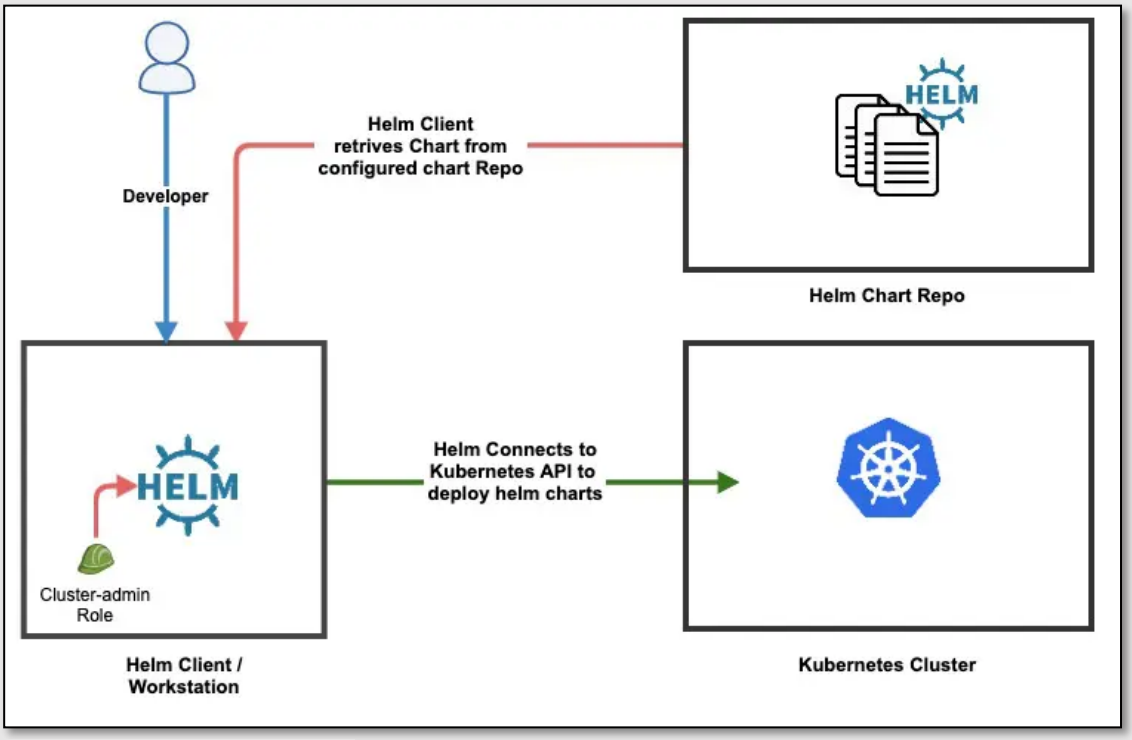


Figura 1 architettura Helm v3.7.1

Helm create

Una volta scaricato l'eseguibile è possibile eseguire l'istruzione per la generazione di un nuovo chart: <PATH_TO_HELM_EXECUTABLE> create <CHART_NAME>

```
C:\Users\workstation\Desktop\helm_guide>helm create my_chart
Creating my_chart
```

Questo comando genera un albero come segue:

```
my_chart
    .helmignore
    Chart.yaml
    values.yaml

    charts
    templates
        deployment.yaml
        hpa.yaml
        ingress.yaml
        NOTES.txt
        service.yaml
        serviceaccount.yaml
        _helpers.tpl

        tests
            test-connection.yaml
```

Come primo passo bisogna creare un nuovo file sotto la cartella templates
con il nome my_template.yaml.
Ne risulterà il seguente albero:

```
my_chart
    .helmignore
    Chart.yaml
    values.yaml

    charts
    templates
        my_template.yaml
```

N.B. per motivi di chiarezza si è scelto di eliminare tutti i file generati da Helm nella cartella templates, nonché di eliminare il contenuto nel file values.yaml.

Helm built-in objects

Helm mette a disposizione degli utenti alcuni elementi noti come oggetti;
essi possono possedere un solo valore, altri oggetti e/o funzioni.
Gli oggetti vengono passati ai template tramite un componente noto come “Template Engine”,
per poi essere utilizzati tramite Go ed in particolare Go template,
libreria che consente la creazione di contenuti di testo dinamici. Quelli built-in sono:

Release: descrive la release stessa e contiene i seguenti oggetti:

- Release.Name
- Release.Namespace
- Release.IsUpgrade
- Release.IsInstall
- Release.Revision
- Release.Service

Values: rappresenta il contenuto espresso nel file values.yaml utilizzato
nei template con la seguente sintassi:
{{ Values.<VAR_KEY> }}

Chart: rappresenta il contenuto espresso nel file Chart.yaml utilizzato
nei template con la seguente sintassi:
{{ Chart.<VAR_KEY> }}

Files: rappresenta l’oggetto messo a disposizione da Helm per accedere a tutti i file all’interno del nostro chart, unica limitazione non è possibile accedere ai templates.
Le funzioni che implementa sono le seguenti:

- Files.Get: fornisce il contenuto di un file sotto forma di stringa in base al nome fornito come parametro. Esempio: Files.Get "< PATH_TO_FILES/FILE_NAME>".
- Files.GetBytes: fornisce il contenuto di un file sotto forma di array di bites, particolarmente utile in caso di file come immagini.
- Files.Glob: fornisce una lista iterabile di files i quali nomi corrispondono ad un pattern fornito come parametro. Esempio: Files.Glob “<PATH_TO_FILES>/*”.
- Files.Lines: fornisce un file sotto forma di array di righe, utile per iterarne le singole righe.
- Files.AsSecrets: fornisce il contenuto di un file codificato in base64 utile per creare dei secrets.
- Files.AsConfig: fornisce il contenuto di un file sotto forma di mappa yaml.

N.B. Quando si parla di nome del file si intende l’intero percorso a partire dalla directory root, in questo caso “my_chart”.

Template: rappresenta l’oggetto Template e fornisce meta informazioni.
In questo caso a fronte del template my_template possiamo ottenere i seguenti dati:

- Template.Name = my_chart/templates/my_template.yaml
- Template.BasePath = my_chart/templates

N.B. tutti gli oggetti built-in iniziano con la maiuscola.

Release

Modifichiamo il file my_template.yaml come segue:

```
#Release object
releaseName: {{ .Release.Name }}
releaseNamespace: {{ .Release.Namespace }}
releaseIsUpgrade: {{ .Release.IsUpgrade }}
releaseIsInstall: {{ .Release.IsInstall }}
releaseVersion: {{ .Release.Revision }}
releaseService: {{ .Release.Service }}
```

N.B La sintassi, comprensiva degli spazi dopo le graffe di apertura e di quelli prima delle graffe di chiusura, è particolarmente stringente ed importante.

Lanciamo la seguente **istruzione di debug** da riga comando posizionandoci al livello superiore rispetto alla cartella my_chart:

```
helm template --dry-run --debug ./my_chart > debug.yaml
```

Questa istruzione genera un file chiamato debug.yaml nel quale viene scritto l'output del my_template.yaml.
In questo caso:

```
---
# Source: my_chart/templates/my_template.yaml
#Release object
releaseName: RELEASE-NAME
releaseNamespace: myproject
releaseIsUpgrade: false
releaseIsInstall: true
releaseVersion: 1
releaseService: Helm
```

N.B. Le variabili di questo oggetto, specialmente name e namespace, vengono impostate con valori di default sovrascrivibili all'atto della creazione del chart e/o della sua installazione via riga di comando.

Values

Modifichiamo il file values.yaml come segue:

```
stringVal: “my-string_2.0”
boolVal: true
intVal: 9
firstFloatVal: 2.09
secondFloatVal: 2.90
thirdFloatVal: 2.00
```

Modifichiamo il file my_template.yaml come segue:

```
#Values object
string: {{ .Values.stringVal }}
boolean: {{ .Values.boolVal }}
integer: {{ .Values.intVal }}
firstFloat: {{ .Values.firstFloatVal }}
secondFloat: {{ .Values.secondFloatVal }}
thirdFloat: {{ .Values.thirdFloatVal }}
```

L’istruzione di debug genererà un output come segue:

```
---
# Source: my_chart/templates/my_template.yaml
#Values object
string: my-string_2.0
boolean: true
integer: 9
firstFloat: 2.09
secondFloat: 2.9
thirdFloat: 2
```

N.B. in questo esempio ci siamo limitati a scrivere valori di tipo numerici, letterali e booleani. Risulta possibile esprimere anche oggetti in formato yaml nonché liste del medesimo formato, per poi essere analizzati tramite apposite funzioni enumerate più avanti.
Si noti il comportamento assunto in fase di analisi nei confronti dei valori numerici con decimali. In alcuni casi risulta conveniente esprimerli tra apici sotto forma di stringhe.

Chart

Modifichiamo il file my_template.yaml come segue:

```
#Chart object
name: {{ .Chart.Name }}
description: {{ .Chart.Description }}
version: {{ .Chart.Version }}
appVersion: {{ .Chart.AppVersion }}
```

L'istruzione di debug genererà un output come segue:

```yaml
---
# Source: my_chart/templates/my_template.yaml
#Chart object
name: my_chart
description: A Helm chart for Kubernetes
type: application
version: 0.1.0
appVersion: 1.16.0
```

N.B. I valori vengono estrapolati direttamente dal file Chart.yaml così composto:

```yaml
apiVersion: v2
name: my_chart
description: A Helm chart for Kubernetes

# A chart can be either an 'application' or a 'library' chart.
#
# Application charts are a collection of templates that can be packaged into versioned archives
# to be deployed.
#
# Library charts provide useful utilities or functions for the chart developer. They're included as
# a dependency of application charts to inject those utilities and functions into the rendering
# pipeline. Library charts do not define any templates and therefore cannot be deployed.
type: application

# This is the chart version. This version number should be incremented each time you make changes
# to the chart and its templates, including the app version.
# Versions are expected to follow Semantic Versioning (https://semver.org/)
version: 0.1.0

# This is the version number of the application being deployed. This version number should be
# incremented each time you make changes to the application. Versions are not expected to
# follow Semantic Versioning. They should reflect the version the application is using.
# It is recommended to use it with quotes.
appVersion: "1.16.0"
```

Files

Files.Get

Creiamo un file sotto la cartella my_chart con nome hello-Files.txt con il seguente contenuto:

```
hello_files
```

Modifichiamo il file my_template.yaml come segue:

```
#Files object
hello_file: {{ .Files.Get “hello-Files.txt” }}
```

L’istruzione di debug genererà un output come segue:

```
---
# Source: my_chart/templates/my_template.yaml
#Files object
hello_file: hello_files
```

Files.GetBytes

Modifichiamo il file my_template.yaml, come segue:

```
#Files object
hello_file: {{ .Files.GetBytes “hello-Files.txt” }}
```

L’istruzione di debug genererà un output come segue:

```
---
# Source: my_chart/templates/my_template.yaml
#Files object
hello_file: [104 101 108 108 111 95 102 105 108 101 115]
```

Files.AsConfig

Modifichiamo il file my_template.yaml come segue:

```
#Files object
hello_file: {{- (.Files.Glob “hello-Files.txt”).AsConfig  | nindent 2 }}
```

L’istruzione di debug genererà un output come segue:

```
---
# Source: my_chart/templates/my_template.yaml
#Files object
hello_file:
  hello-Files.txt: hello_files
```

N.B. L’utilizzo della funzione “nindent” non è essenziale.
Avremmo potuto indentare a mano. Vedi esempio sotto.

Files.AsSecrets

Modifichiamo il file my_template.yaml come segue:

```
#Files object
hello_file:
 {{ (.Files.Glob “hello-Files.txt”).AsSecrets }}
```

L’istruzione di debug genererà un output come segue:

```
---
# Source: my_chart/templates/my_template.yaml
#Files object
hello_file:
  hello-Files.txt: aGVsbG9fZmlsZXMNCg0KDQoNCg0K
```

N.B. Si noti che in questo caso la sintassi è diversa, la funzione “AsSecret” così come “AsConfig” può essere chiamata solo su oggetti di tipo File.
Ne deriva un output chiave-valore dove per chiave otteniamo il nome del file (comprensivo del percorso se non presente al primo livello) e per valore il suo contenuto codificato in base64.
In questo caso per mantenere la compatibilità con il formato yaml dobbiamo andare a capo e dare due spazi di indentazione prima di poter chiamare la funzione di cui sopra.

Controllo del flusso

Con Helm e più in particolare con il supporto fornito dalla libreria Go Template si ha la possibilità di esprimere strutture di controllo note come "actions". Le più rilevanti sono:

If - Else If - Else

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

Una pipeline è valutata false se il suo valore risulta:

- False (booleano)
- Zero (numerico)
- Vuoto o Null (stringhe)
- Vuoto (collection)

In tutti gli altri casi la pipeline sarà valutata come true. Esempio:

Modifichiamo il file my_template.yaml come segue:

```
#Actions
hello_actions:
{{- if .Values.boolVal }}
  boolVal: true
{{- else }}
  boolVal: false
{{- end }}
{{- if eq .Values.stringVal "my-string_2.0" }}
  stringVal: 'true'
{{- else }}stringVal: 'false'
{{- end }}
```

Osserviamone l'output con l'istruzione di debug.

N.B Ogni If/Else è delimitato con {{ end }}.
Il simbolo "-" serve ad eliminare gli spazi creati tramite l'inserimento delle istruzioni, particolarmente utile in fase di definizione di un file yaml via Helm.

Range

L'operatore range rappresenta il ciclo for, ci consente pertanto di iterare su elementi di una lista così come su dei files.

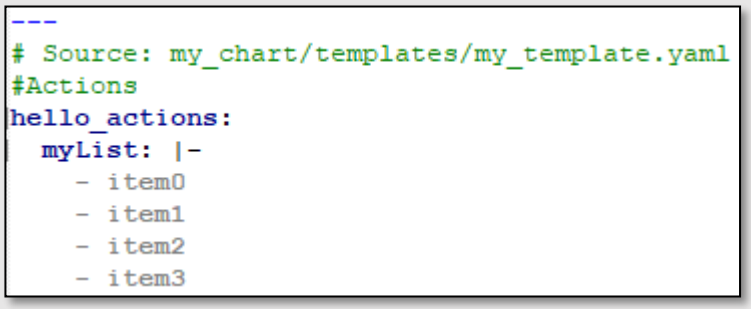
Modifichiamo il values.yaml, aggiungendo una lista (o slice), come segue:

```
myList:
  - item0
  - item1
  - item2
  - item3
```

Modifichiamo il my_template.yaml come segue:

```
#Actions
hello_actions:
  myList: |-
    {{- range .Values.myList }}
    - {{ . }}
    {{- end }}
```

L'output atteso dopo aver lanciato l'istruzione di debug è il seguente:



N.B A seguito dell'operatore "range" viene fornito un elemento iterabile.
In questo caso l'elemento iEsimo è rappresentato dal "." ed anche questa istruzione come l'If/Else termina con {{ end }}.

Funzioni

In questo capitolo verranno enumerate le principali funzioni fornite da Helm e più in generale da Go Template nonché Sprig. Per una lista più dettagliata si rimanda ai seguenti indirizzi:

- https://helm.sh/docs/chart_template_guide/functions_and_pipelines/
- https://godoc.org/text/template
- https://masterminds.github.io/sprig/

Prima di iniziare risulta utile chiarire, per chi non ne fosse già a conoscenza, il termine pipeline. Come da documentazione: "*una pipeline è un concetto ripreso da Unix e rappresenta un tool per aggregare fra loro diversi comandi, atti ad esprimere in maniera compatta una serie di trasformazioni.*" Una pipeline prevede l'utilizzo del seguente simbolo: "|".

String functions

Dato lo scopo di Helm (creare e gestire risorse Kubernetes)
esse risultano senz'altro le più utili ed utilizzate.

Quote

Dato il values.yaml possiamo modificare il file my_template.yaml come segue:

```
#Functions
string_functions:
  quote: {{ .Values.stringVal | quote }}
```

Lanciando l'istruzione di debug l'output atteso è il seguente:

```
---
# Source: my_chart/templates/my_template.yaml
#Actions
string_functions:
  quote: "my-string_2.9"
```

N.B. Rispetto a quanto fatto nel capitolo relativo ai values in questo caso vengono aggiunti dei doppi apici al valore analizzato; questa funzione risulta particolarmente utile
quando si lavora con stringhe in formato Json.

Upper

Dato il values.yaml possiamo modificare il file my_template.yaml come segue:

```
#Functions
string_functions:
  quote: {{ .Values.stringVal | quote }}
  quoteAndUpper: {{ .Values.stringVal | quote | upper }}
```

Lanciando l'istruzione di debug l'output atteso è il seguente:

```
---
# Source: my_chart/templates/my_template.yaml
#Functions
string_functions:
  quote: "my-string_ 2.9"
  quoteAndUpper: "MY-STRING_ 2.9"
```

N.B. Si noti l'utilizzo di due pipeline utili nella concatenazione di più operazioni atte ad ottenere il risultato finale atteso. La funzione "upper" così come quella "quote" possono essere utilizzate singolarmente.

Title

Dato il values.yaml possiamo modificare il file my_template.yaml come segue:

```
#Functions
string_functions:
  quote: {{ .Values.stringVal | quote }}
  quoteAndUpper: {{ .Values.stringVal | quote | upper }}
  title: {{ .Values.stringVal | title }}
```

Lanciando l'istruzione di debug l'output atteso è il seguente:

```
---
# Source: my_chart/templates/my_template.yaml
#Functions
string_functions:
  quote: "my-string_ 2.9"
  quoteAndUpper: "MY-STRING_ 2.9"
  title: My-String_ 2.9
```

Squote

Dato il values.yaml possiamo modificare il file my_template.yaml come segue:

```
#Functions
string_functions:
  quote: {{ .Values.stringVal | quote }}
  quoteAndUpper: {{ .Values.stringVal | quote | upper }}
  title: {{ .Values.stringVal | title }}
  squote: {{ .Values.stringVal | squote }}
```

Lanciando l'istruzione di debug l'output atteso è il seguente:

```
---
# Source: my_chart/templates/my_template.yaml
#Functions
string_functions:
  quote: "my-string_ 2.9"
  quoteAndUpper: "MY-STRING_ 2.9"
  title: My-String_ 2.9
  squote: 'my-string_ 2.9'
```

Indent

Dato il values.yaml possiamo modificare il file my_template.yaml come segue:

```
#Functions
string_functions:
  quote: {{ .Values.stringVal | quote }}
  quoteAndUpper: {{ .Values.stringVal | quote | upper }}
  title: {{ .Values.stringVal | title }}
  squote: {{ .Values.stringVal | squote }}
  indent:{{ .Values.stringVal | indent 1 }}
```

Lanciando l'istruzione di debug l'output atteso è il seguente:

```
---
# Source: my_chart/templates/my_template.yaml
#Functions
string_functions:
  quote: "my-string_ 2.9"
  quoteAndUpper: "MY-STRING_ 2.9"
  title: My-String_ 2.9
  squote: 'my-string_ 2.9'
  indent:  my-string_ 2.9
```

Nindent

Dato il values.yaml possiamo modificare il file my_template.yaml come segue:

```
#Functions
string_functions:
  quote: {{ .Values.stringVal | quote }}
  quoteAndUpper: {{ .Values.stringVal | quote | upper }}
  title: {{ .Values.stringVal | title }}
  squote: {{ .Values.stringVal | squote }}
  indent:{{ .Values.stringVal | indent 1 }}
  nindent:{{- .Values.stringVal | nindent 4 }}: helloNindent
```

Lanciando l'istruzione di debug l'output atteso è il seguente:

```
---
# Source: my_chart/templates/my_template.yaml
#Functions
string_functions:
  quote: "my-string_ 2.9"
  quoteAndUpper: "MY-STRING_ 2.9"
  title: My-String_ 2.9
  squote: 'my-string_ 2.9'
  indent:  my-string_ 2.9
  nindent:
    my-string_ 2.9: helloNindent
```

Replace

Dato il values.yaml possiamo modificare il file my_template.yaml come segue:

```
#Functions
string_functions:
  quote: {{ .Values.stringVal | quote }}
  quoteAndUpper: {{ .Values.stringVal | quote | upper }}
  title: {{ .Values.stringVal | title }}
  squote: {{ .Values.stringVal | squote }}
  indent:{{ .Values.stringVal | indent 1 }}
  nindent:{{- .Values.stringVal | nindent 4 }}: helloNindent
  replace: {{- .Values.stringVal | indent 1 | replace "-" "_" }}
```

Lanciando l'istruzione di debug l'output atteso è il seguente:

```
---
# Source: my_chart/templates/my_template.yaml
#Functions
string_functions:
  quote: "my-string_ 2.9"
  quoteAndUpper: "MY-STRING_ 2.9"
  title: My-String_ 2.9
  squote: 'my-string_ 2.9'
  indent:  my-string_ 2.9
  nindent:
    my-string_ 2.9: helloNindent
  replace: my_string_ 2.9
```

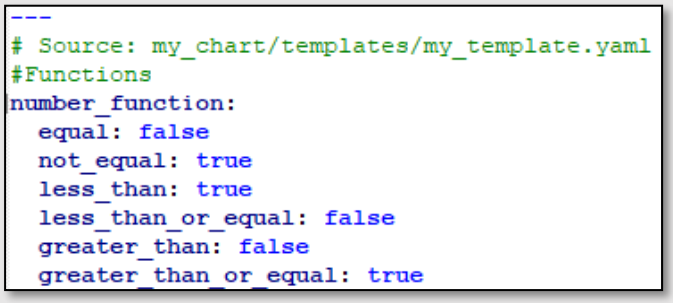
Number functions

In questo paragrafo verranno esplicate le principali funzioni numeriche:
“==”, “!=”, “<”, “<=”, “>”, “>=”.
Per ulteriori dettagli si rimanda ai link di cui sopra.

Dato il values.yaml possiamo modificare il file my_template.yaml come segue:

```
#Functions
number_functions:
  equal: {{ (eq .Values.intVal 9.1) }}
  not_equal: {{ (ne .Values.intVal 9.1) }}
  less_than: {{- (lt .Values.intVal 18.0) | toString | indent 1 }}
  less_than_or_equal: {{- (le 18.0 .Values.intVal) | toString | indent 1 }}
  greater_than: {{ (gt .Values.intVal 9.0) }}
  greater_than_or_equal: {{ (ge .Values.intVal 9.0) }}
```

Lanciando l’istruzione di debug l’output atteso è il seguente:



N.B. Si noti la sintassi: <FUNZIONE> <ARG1> <ARG2>; si riporta all’attenzione anche che a fronte di numeri non analizzati via template engine, 9.1 nel primo caso, essi devono essere espressi con tanto di decimali separati da punto, mentre per i valori analizzati via template engine, “.Values.intVal”, non risulta necessario. Ultimo punto di attenzione, riguarda le righe 4 e 5 del file my_template di cui sopra; l’aver voluto indentare il risultato tramite apposita funzione, indent in questo caso, comporta dover prima di tutto trasformare il risultato in una stringa, in quanto tale funzione può essere richiamata solo su valori di questo tipo. Questo è un esempio della flessibilità fornita da Helm e del funzionamento delle pipeline.

Flow control functions

Di seguito andremo ad elencare le principali funzioni riguardanti il controllo del flusso, “and, “or”. Per ulteriori dettagli si rimanda ai link di cui sopra.

Dato il values.yaml possiamo modificare il file my_template.yaml come segue:

```
#Functions
operators:
  and: {{- and (lt .Values.intVal 18.0) (gt 9.0 .Values.intVal) | toString | indent 1 }}
  or: {{- or (eq .Values.stringVal “my_string”) (contains “my” .Values.stringVal) | toString | indent 1 }}
  andOr: {{- or (and (lt .Values.intVal 18.0) (gt 9.0 .Values.intVal)) (ge 9.0 .Values.intVal) | toString | indent 1 }}
  orAnd: {{- and (or ( lt .Values.intVal 8.0) (gt 9.0 .Values.intVal)) (ge 9.0 .Values.intVal) | toString | indent 1 }}
```

Lanciando l’istruzione di debug l’output atteso è il seguente:

```
---
# Source: my_chart/templates/my_template.yaml
#Functions
operators:
  and: false
  or: true
  andOr: true
  orAnd: false
```

Si noti la sintassi sopra utilizzata:

```
<OPERATORE_LOGICO> (<CONDIZIONE1>) (<CONDIZIONE2>)
```

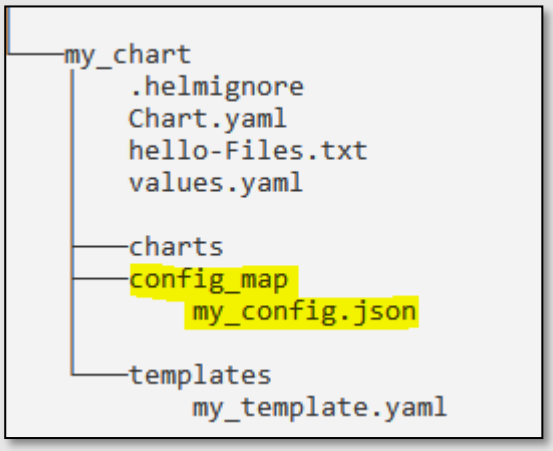
nonché:

```
<OPERATORE_LOGICO> (<OPERATORE_LOGICO> (<CONDIZIONE1>) (<CONDIZIONE2>)) (<CONDIZIONE3>)
```

Esempi pratici

Creare un ConfigMap da un file json

Creiamo una cartella sotto my_chart ed inseriamo dentro un file json:



my_config.json:

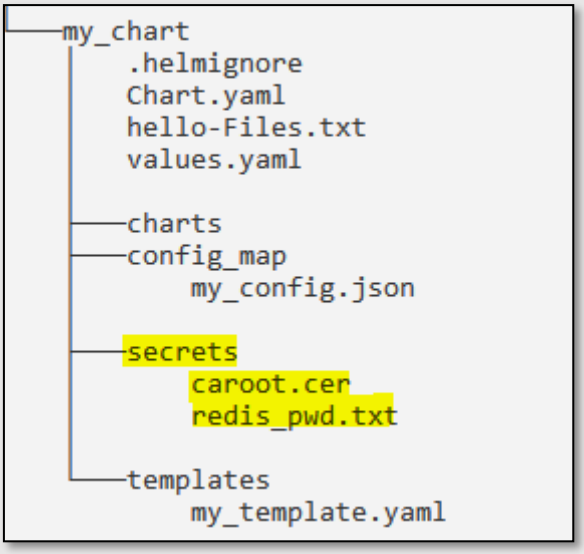
```
{
        “env”: “prod”,
        “variables”: [1,2,3,4,5],
        “fooObj”: {
                “first”: 1,
                “second”: true,
                “third”: 9
        }
}
```

Modifichiamo il file my_template.yaml come segue,
ed osserviamone l’output lanciando l’istruzione di debug:

```
{{- $file := .Files.Get “config_map/my_config.json” }} #Var ref
apiVersion: v1
kind: ConfigMap
data:
  my_config.json: {{- $file | quote | indent 1 }}
```

Creare un Secret con più items

Creiamo una cartella sotto my_chart ed inseriamo dentro due files:



Modifichiamo il file my_template.yaml come segue:

```
apiVersion: v1
kind: Secret
type: Opaque
data:
{{- range $path, $bytes := .Files.Glob “secrets/*” }}
  {{- base $path | nindent 2 }}: >- {{- $bytes | toString | b64enc | nindent 4 }}
{{- end }}
```

L’output atteso lanciando l’istruzione di debug è il seguente:

```
---
# Source: my_chart/templates/my_template.yaml
apiVersion: v1
kind: Secret
type: Opaque
data:
  caroot.cer: >-
    LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tDQpNSU1EU1RDQ0FpMENGSE9iVmJJRG1teFltK0FqUl1
  redis_pwd.txt: >-
    cGFzc3cwcmQ=
```

Fonti

- https://helm.sh/docs/topics/architecture/#:~:text=The%20Helm%20client%20and%20library%20is%20written%20in%20the%20Go%20programming%20language.
- https://devopscube.com/install-configure-helm-kubernetes/ (immagine 1)
- https://go.dev/
- https://pkg.go.dev/text/template
- https://masterminds.github.io/sprig/