Menu

DevOpsHint

Use of Flow Control Statements in Helm with Example

Updated: December 18, 2023 by Prasad Hole

In this article, we are going to cover Use of Flow Control Statements in Helm with Example | flow control structures of helm `if`, `with`, and `range`, in Helm templates.. We'll focus on its templating system, using simple 'if,' 'with,' and 'range' statements to make Kubernetes setups flexible. These constructs allow for dynamic and conditional customization of Kubernetes resources.

Table of Contents

Prerequisites

- AWS Account with Ubuntu 22.04 LTS EC2 Instance
- Minikube and kubectl Installed

Install Minikube and kubectl by following the official documentation for your operating system:

Minikube Installation Guide

Install Minikube on Ubuntu 22.04 LTS

- Helm Installed:

Install Helm by following the official documentation:

Helm Installation Guide

Use of if Statement in Helm with Example

The `if` statement in Helm's templating language allows for conditional logic. It evaluates a condition and includes or excludes specific sections of YAML based on the result. The condition is typically expressed using comparison or equality operators. It checks a condition and executes a block of code if the condition is true.

- The `if` statement can be chained with `else if` and `else` for handling multiple conditions.
- Common comparison operators include `eq`, `ne`, `lt`, `le`, `gt`, `ge`.

Syntax:

```
{{- if condition }}
  # Code to execute if the condition is true
{{- else if anotherCondition }}
  # Code to execute if another condition is true
{{- else }}
```

```
  # Code to execute if none of the conditions are true
{{- end }}
```

- '`{{-`' and '`-}}`' are used for trimming whitespaces.
- `if` is followed by the condition to be evaluated.
- `else if` and `else` are optional, providing additional branches for different conditions.

A condition 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`)

Example:

first we will create a chart named 'helloworld'.

```
helm create helloworld
```

then open the directory in which values.yaml file is present.

```
cd helloworld
```

now lets modify values file. use the following command to modify in it.

```
nano values.yaml
```

modify it as shown below.

```
repicaCount: 3
```

```
envVars:
    enabled: false    # Change to true to enable envVars
    valueOne: "first_value"
    valueTwo: "second_value"
```

```yaml
replicaCount: 3

image:
  repository: nginx
  pullPolicy: IfNotPresent
  # Overrides the image tag whose default is the chart appVersion.
  tag: ""

envVars:
  enabled: false  # Change to true to enable envVars
  valueOne: "first_value"
  valueTwo: "second_value"
```

to save the modification press ctrl+x, shift+y and enter.

then we will modify the deployment.yaml file,

first open the directory in which the file present,

```
cd templates
```

then use following command to modify in deployment.yaml

```
nano deployment.yaml
```

now modify the part as shown in below image. Here env section is added using if statement inside the container section.

```
{{- if .Values.envVars.enabled }}
env:
    - name: ENV_VAR_ONE
      value: {{ .Values.envVars.valueOne | quote }}
    - name: ENV_VAR_TWO
      value: {{ .Values.envVars.valueTwo | quote }}
{{- else }}
env:
    - name: ENV_VAR_DEFAULT
      value: "default_value"
{{- end }}
```

```
containers:
  - name: {{ .Chart.Name }}
    securityContext:
      {{- toYaml .Values.securityContext | nindent 12 }}
    image: "{{ .Values.image.repository }}:{{ .Values.image.tag | default .Chart.AppVersion }}"
    imagePullPolicy: {{ .Values.image.pullPolicy }}
    ports:
      - name: http
        containerPort: {{ .Values.service.port }}
        protocol: TCP
    {{- if .Values.envVars.enabled }}
    env:
      - name: ENV_VAR_ONE
        value: {{ .Values.envVars.valueOne | quote }}
      - name: ENV_VAR_TWO
        value: {{ .Values.envVars.valueTwo | quote }}
    {{- else }}
    env:
      - name: ENV_VAR_DEFAULT
        value: "default_value"
    {{- end }}
    livenessProbe:
      httpGet:
        path: /
        port: http
```

to save the modification press ctrl+x, shift+y and enter.

In this example:

- The `.Values.envVars.enabled` checks if `envVars` is enabled in the Helm values file.
- If it's set to `true`, it includes the `env` section with the specified environment variables.
- If it's set to `false` or not defined, it includes a default environment variable (`ENV_VAR_DEFAULT`) with a default value.

now exit the directory using following command.

```
cd
```

run the template command to see if modifications are done.

```
helm template helloworld
```

Output:

```
# Source: helloworld/templates/deployment.yaml

apiVersion: apps/v1
kind: Deployment
metadata:
name: release-name-helloworld
labels:
helm.sh/chart: helloworld-0.1.0
app.kubernetes.io/name: helloworld
app.kubernetes.io/instance: release-name
app.kubernetes.io/version: "1.16.0"
app.kubernetes.io/managed-by: Helm
spec:
replicas: 3
selector:
matchLabels:
app.kubernetes.io/name: helloworld
app.kubernetes.io/instance: release-name
template:
metadata:
```

```
labels:
helm.sh/chart: helloworld-0.1.0
app.kubernetes.io/name: helloworld
app.kubernetes.io/instance: release-name
app.kubernetes.io/version: "1.16.0"
app.kubernetes.io/managed-by: Helm
spec:
serviceAccountName: release-name-helloworld
securityContext:
{}
containers:
- name: helloworld
securityContext:
{}
image: "nginx:1.16.0"
imagePullPolicy: IfNotPresent
ports:
- name: http
containerPort: 80
protocol: TCP
env:
- name: ENV_VAR_DEFAULT
value: "default_value"
livenessProbe:
httpGet:
path: /
port: http
readinessProbe:
httpGet:
path: /
port: http
resources:
{}
```

Since, envVars is set to enabled: false so instead of first_value and second_value, default value is printed.

Now lets change the enabled: false to true. Follow the same procedure to modify values.yaml file and set the envVars to enabled:true

```
envVars:
    enabled: true    # Change to true to enable envVars
    valueOne: "first_value"
    valueTwo: "second_value"
```

Now run the helm template helloworld command again.

Output:

```
ports:
  - name: http
    containerPort: 80
    protocol: TCP
env:
  - name: ENV_VAR_ONE
    value: "first_value"
  - name: ENV_VAR_TWO
    value: "second_value"
livenessProbe:
  httpGet:
    path: /
    port: http
```

as you can see in env section printed the name and value of first and second variable.

Use of with Statement in Helm with example:

The `with` statement simplifies referencing nested structures in Helm templates. It sets a context for a block of code, reducing redundancy and improving code readability. This is particularly handy when dealing with complex configurations and nested values within the Helm chart.

- The `with` statement sets the context for the block, allowing direct access to the fields within `.Values`.

- Useful when dealing with nested structures to avoid repetitive references.
- The . (dot) is a special identifier in Helm that represents the current context or scope. When used with `with`, it allows you to reference values from the current context within the new scope.

Syntax:

```
{{- with context }}
  # Code to execute within the specified context
{{- end }}
```

- '`{{-`' and '`-}}`' trim whitespaces.
- `with` is followed by the context or variable, setting the scope for the enclosed block of code.

Example:

Follow the same procedure mentioned above till modifying values.yaml.

add the envVariables part in values.yaml file.

```
service:
  type: ClusterIP
  port: 80

envVariables:
 - name: DATABASE_URL
   value: "your-database-url"
 - name: API_KEY
```

```
    value: "your-api-key"
  - name: DEBUG_MODE
    value: "true"

ingress:
  enabled: false
  className: ""
  annotations: {}
    # kubernetes.io/ingress.class: nginx
    # kubernetes.io/tls-acme: "true"
  hosts:
    - host: chart-example.local
      paths:
        - path: /
          pathType: ImplementationSpecific
```

to save the modification press ctrl+x, shift+y and enter.

then open the templates directory and modify the deployment.yaml file

```
cd templates
```

```
nano deployment.yaml
```

We will use ‘with’ at container block after port part.

```
      containers:
        - name: {{ .Chart.Name }}
          securityContext:
            {{- toYaml .Values.securityContext | nindent 12 }}
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag |
default .Chart.AppVersion }}"
          imagePullPolicy: {{ .Values.image.pullPolicy }}
          ports:
            - name: http
              containerPort: {{ .Values.service.port }}
              protocol: TCP
          {{- with .Values.envVariables }}
          env:
      {{- range . }}
      - name: {{ .name }}
```

```
          value: {{ .value | quote }}
        {{- end }}
            {{- end }}
            livenessProbe:
              httpGet:
                path: /
                port: http
```

to save the modification press ctrl+x, shift+y and enter.

exit the directories.

```
cd
```

then run the helm template command

```
helm template helloworld
```

Output:

```
# Source: helloworld/templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: release-name-helloworld
  labels:
    helm.sh/chart: helloworld-0.1.0
    app.kubernetes.io/name: helloworld
    app.kubernetes.io/instance: release-name
    app.kubernetes.io/version: "1.16.0"
    app.kubernetes.io/managed-by: Helm
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: helloworld
      app.kubernetes.io/instance: release-name
  template:
    metadata:
      labels:
        helm.sh/chart: helloworld-0.1.0
        app.kubernetes.io/name: helloworld
```

```
        app.kubernetes.io/instance: release-name
        app.kubernetes.io/version: "1.16.0"
        app.kubernetes.io/managed-by: Helm
    spec:
      serviceAccountName: release-name-helloworld
      securityContext:
        {}
      containers:
        - name: helloworld
          securityContext:
            {}
          image: "nginx:1.16.0"
          imagePullPolicy: IfNotPresent
          ports:
            - name: http
              containerPort: 80
              protocol: TCP
          env:
        - name: DATABASE_URL
          value: "your-database-url"
        - name: API_KEY
          value: "your-api-key"
        - name: DEBUG_MODE
          value: "true"
          livenessProbe:
            httpGet:
              path: /
              port: http
          readinessProbe:
            httpGet:
              path: /
              port: http
          resources:
            {}
```

To align your Helm values file with the modified `deployment.yaml` using the `with` statement, you need to structure your values file accordingly.

Use of range Statement with example:

The `range` statement enables iteration over lists or maps in Helm templates. It's a powerful tool for dynamically generating Kubernetes resources based on a list of values. This feature is

crucial for creating scalable charts that can adapt to varying numbers of components or services.

- The `range` statement iterates over the list or map provided.
- It is valuable when generating resources dynamically based on input values.

Syntax:

```
{{- range iterable }}
  # Code to execute for each item in the iterable
{{- end }}
```

- '`{{-`' and '`-}}`' trim whitespaces.
- `range` is followed by the iterable (e.g., a list or map).
- The block of code inside the `range` is executed for each item in the iterable.

Example:

Follow the same steps as if statement until modifying values.yaml file

Modify the values file using following command.

```
nano values.yaml
```

add the `envVariable` section in it.

```
envVariables:
    - name: DB_HOST
      value: "localhost"
    - name: DB_PORT
      value: "5432"
    - name: DB_KEY
      value: "secret_key"
```

```
serviceAccount:
  # Specifies whether a service account should be created
  create: true
  # Automatically mount a ServiceAccount's API credentials?
  automount: true
  # Annotations to add to the service account
  annotations: {}
  # The name of the service account to use.
  # If not set and create is true, a name is generated using the fullname template
  name: ""

envVariables:
  - name: DB_HOST
    value: "localhost"
  - name: DB_PORT
    value: "5432"
  - name: API_KEY
    value: "secret_key"

podAnnotations: {}
podLabels: {}
```

to save the modification press ctrl+x, shift+y and enter.

Now open the templates directory

```
cd templates
```

Then modify the deployment.yaml file.

```
nano deployment.yaml
```

add env section in container section as shown below.

```
env:
  {{- range .Values.envVariables }}
  - name: {{ .name }}
    value: {{ .value | quote }}
  {{- end }}
```

```
containers:
  - name: {{ .Chart.Name }}
    securityContext:
      {{- toYaml .Values.securityContext | nindent 12 }}
    image: "{{ .Values.image.repository }}:{{ .Values.image.tag | default .Chart.AppVersion }}"
    imagePullPolicy: {{ .Values.image.pullPolicy }}
    env:
     {{- range .Values.envVariables }}
     - name: {{ .name }}
       value: {{ .value | quote }}
     {{- end }}
    ports:
      - name: http
        containerPort: {{ .Values.service.port }}
        protocol: TCP
    livenessProbe:
      httpGet:
        path: /
        port: http
```

to save the modification press ctrl+x, shift+y and enter.

exit the directories.

```
cd
```

Run the helm template command.

```
helm template helloworld
```

Output:

```yaml
# Source: helloworld/templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: release-name-helloworld
  labels:
    helm.sh/chart: helloworld-0.1.0
    app.kubernetes.io/name: helloworld
    app.kubernetes.io/instance: release-name
    app.kubernetes.io/version: "1.16.0"
    app.kubernetes.io/managed-by: Helm
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: helloworld
      app.kubernetes.io/instance: release-name
  template:
    metadata:
      labels:
        helm.sh/chart: helloworld-0.1.0
        app.kubernetes.io/name: helloworld
        app.kubernetes.io/instance: release-name
        app.kubernetes.io/version: "1.16.0"
        app.kubernetes.io/managed-by: Helm
    spec:
      serviceAccountName: release-name-helloworld
      securityContext:
        {}
      containers:
        - name: helloworld
          securityContext:
            {}
          image: "nginx:1.16.0"
          imagePullPolicy: IfNotPresent
          env:
           - name: DB_HOST
             value: "localhost"
           - name: DB_PORT
             value: "5432"
           - name: API_KEY
             value: "secret_key"
          ports:
            - name: http
              containerPort: 80
              protocol: TCP
          livenessProbe:
            httpGet:
              path: /
```

```
    port: http
readinessProbe:
  httpGet:
    path: /
    port: http
resources:
  {}
```

You can see the env section in the output is printed.

- The `range` statement is used to iterate over the list of environment variables defined in `.Values.envVariables`.
- For each variable in the list, a new block is created in the `spec.template.spec.containers.env` section of the `Deployment`.
- The template within the `range` block accesses properties of each environment variable such as `name` and `value`.

Importance Flow Control Statements in Helm

The importance of flow controls (`if`, `with`, and `range`) in Helm cannot be overstated. Here are key reasons why they are crucial in Helm:

1. **Dynamic Configurations:**
   - `if` statements enable conditional adjustments, allowing dynamic configurations based on different environments.
2. **Reduced Redundancy with `with`:**
   - The `with` statement simplifies referencing nested structures, reducing redundancy and enhancing code readability.
3. **Iterative Resource Generation with `range`:**

- `range` statements facilitate the dynamic creation of resources, essential for scalable and adaptable Helm charts.

4. **Adaptability and Readability:**
    - Helm flow controls enhance adaptability by adjusting configurations, promoting modularity, and making charts more readable in diverse deployment scenarios.

Best Practices of Flow Control Statements in Helm

Here are the main best practices for using flow controls in Helm charts:

1. **Whitespace Trimming:**
    - Use '`{{-`' and '`-}}`' to trim unnecessary whitespaces, ensuring cleaner YAML output.
2. **Consistent Indentation:**
    - Maintain consistent indentation for a well-organized and readable code structure.
3. **Commenting:**
    - Add comments to explain the purpose of flow controls and any complex logic for better understanding.
4. **Use Helper Functions for Complex Logic:**
    - Consider moving intricate or reusable logic to helper functions in `helpers.tpl` to keep main templates focused.
5. **Avoid Nested Flow Controls When Possible:**

- Limit the depth of nested flow controls to maintain simplicity and ease of management.

6. **Error Handling:**
   - Use `{{- fail "message" }}` in `if` statements for explicit error handling to avoid silent failures.
7. **Group Related Logic:**
   - Group related logic together to improve clarity and separate different concerns into distinct sections.
8. **Avoid Hardcoding Values:**
   - Refrain from hardcoding values directly within templates; rely on values from `values.yaml` or external configurations.
9. **Testing and Validation:**
   - Implement thorough testing of Helm charts, including scenarios involving different flow control paths.
10. **Documentation:**
    - Maintain documentation outlining the purpose and usage of flow controls within Helm charts.

These practices collectively contribute to creating maintainable, readable, and error-resistant Helm charts with effective use of flow controls.

Conclusion:

In conclusion, Use of Flow Control Statements in Helm with Example | using Helm charts with flow controls like `if`, `with`, and `range`

involves keeping things tidy, using helper functions smartly, and simplifying without making it too complicated. It's about organizing, testing, and documenting for practical use in different situations. It ensures that your Helm charts are well-structured, adaptable, and straightforward for seamless use in diverse environments.

Related Articles:

How to use helm lint, helm –debug –dry-run and helm template

About Prasad Hole

Leave a Comment

Name *

Email *

Website

Save my name, email, and website in this browser for the next time I comment.

Post Comment

This site uses Akismet to reduce spam. Learn how your comment data is processed.

Search

Top 5 Beaches Near Pune to visit within 200 km

Monitor Docker Containers with Prometheus and Grafana

Cloudways Autonomous-Kubernetes Hosting for High Traffic Site

4 Types of Elastic Load Balancer in AWS

Deploy Application to AWS Elastic Beanstalk

Pull Image from DockerHub Private Registry using Helm in Kubernetes

Terraform Cloud Sentinel Policy and Remote Backends

About DevOpsHint

DevOpsHint is a Community site where you can find about How to Guides, Articles and Troubleshooting Tips for Various current DevOps,GitOps,DevSecOps,SRE Tools and Resources.

  

DevOps Resources

Free DevOps Resources

Consulting and Support

Terraform with AWS Consulting and Job Support

Prometheus Consulting and Job Support

Site Links

About Us

Contact Us

Privacy Policy

Terms and Conditions