Пакетный менеджер Helm. Конфигурация и развертывание пакетов


HELM

Why helm?

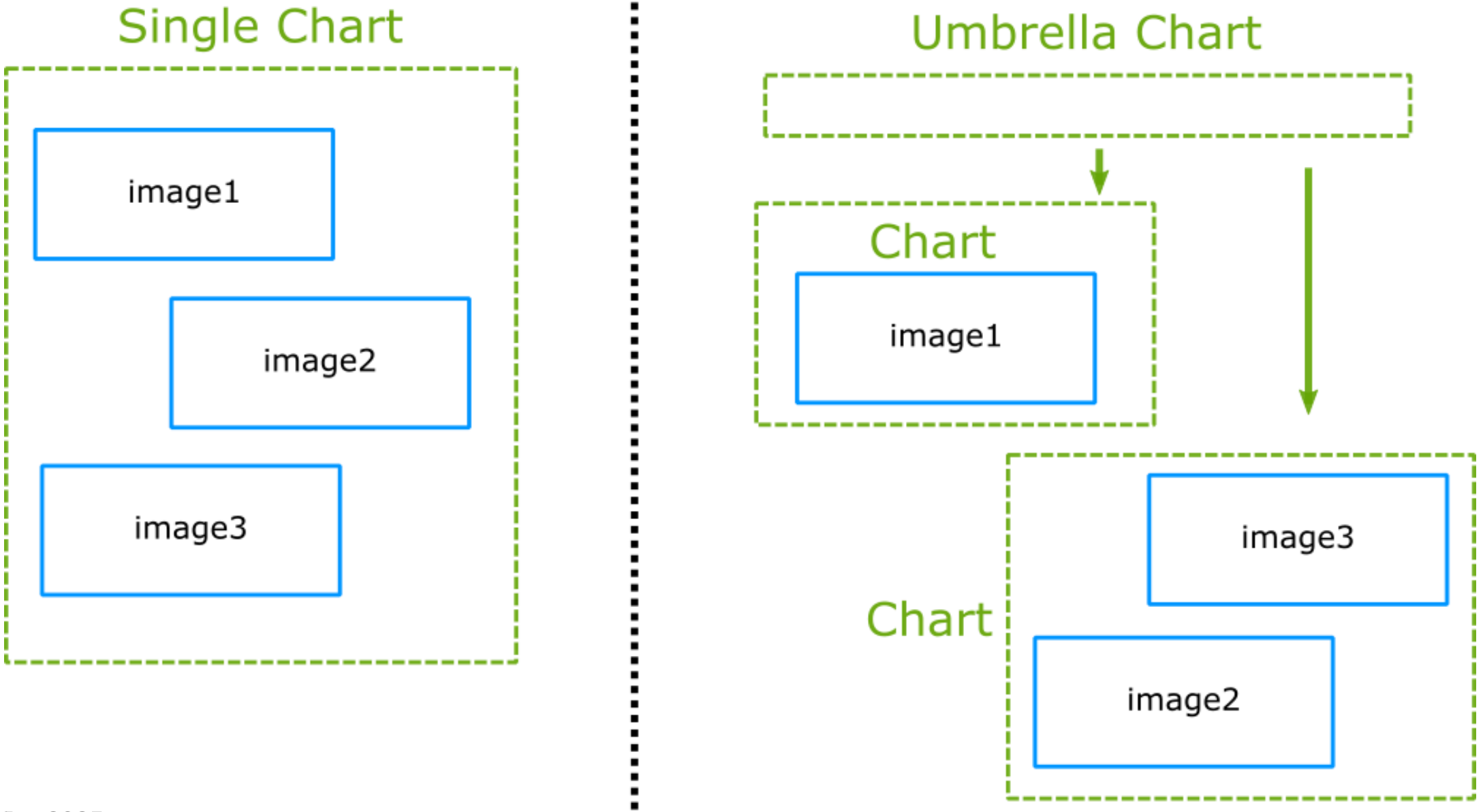
- Grouping related Kubernetes manifests in a single entity (the chart)
- Basic templating and values for Kubernetes manifests
- Dependency declaration between applications (chart of charts)
- A registry of available applications to be deployed (Helm repository)
- A view of a Kubernetes cluster at the application/chart level
- Managing of chart installation/upgrades as a whole
- Built-in rollback of a chart to a previous version without running a CI/CD pipeline again

Struture

Helm can create the chart structure in a single command line:

```
helm create nginx
```

```
.
├nginx
├── Chart.yaml  # A YAML file containing information about the chart
├── README.md   # OPTIONAL: A human-readable README file
├── LICENSE     # OPTIONAL: A plain text file containing the license for the chart
├── crds/       # Custom Resource Definitions
├── charts      # A directory containing any charts upon which this chart depends.
├── templates   # A directory of templates that, when combined with values,
│               # will generate valid Kubernetes manifest files.
│   ├── Note.txt # OPTIONAL: A plain text file containing short usage Note
│   ├── _helpers.tpl # file will have the re-usable component throughout the chart.
│   ├── deployment.yaml
│   ├── ingress.yaml
│   └── service.yaml
└── values.yaml # The default configuration values for this chart
```


Single Chart
image1
image2
image3
Umbrella Chart
Chart
image1
Chart
image3
image2

Use Subcharts (Umbrella charts) to Manage Your Dependencies

The folder structure should be in the following order:

```
backend-chart
  - Chart.yaml
  - charts
      - message-queue
          - Chart.yaml
          - templates
          - values.yaml
      - database
          - Chart.yaml
          - templates
          - values.yaml
  - values.yaml
```

Chart.yaml

the **chart.yaml** in the parent chart should list any dependencies and conditions:

```yaml
apiVersion: v2
name: backend-chart
description: A Helm chart for backend

dependencies:
  - name: apache
    version: 1.2.3
    repository: http://example.com/charts
    alias: new-subchart-1
    condition: subchart1.enabled, global.subchart1.enabled
      tags:
        - front-end
        - subchart1
```

values.yaml

you can set or override the values of subcharts in the parent chart with the following values.yaml file

```yaml
message-queue:
  enabled: true
  image:
    repository: acme/rabbitmq
    tag: latest
database:
  enabled: false
```

Useful Commands for Debugging Helm Charts

helm lint: The linter tool conducts a series of tests to ensure your chart is correctly formed.
helm install --dry-run --debug: This function renders the templates and shows the resulting resource manifests.
helm get manifest: This command retrieves the manifests of the resources that are installed to the cluster.
helm get values: This command is used to retrieve the release values installed to the cluster.

Operators are functions

and

Returns the boolean AND of two or more arguments (the first empty argument, or the last argument).

```
and .Arg1 .Arg2
```

or

Returns the boolean OR of two or more arguments (the first non-empty argument, or the last argument).

```
or .Arg1 .Arg2
```

not

Returns the boolean negation of its argument.

Operators are functions

eq

Returns the boolean equality of the arguments (e.g., Arg1 == Arg2).

```
eq .Arg1 .Arg2
```

ne

Returns the boolean inequality of the arguments (e.g., Arg1 != Arg2)

```
ne .Arg1 .Arg2
```

> For templates, the operators (eq, ne, lt, gt, and, or and so on) are all implemented as functions. In pipelines, operations can be grouped with parentheses ((, and)).

Operators are functions

lt

Returns a boolean true if the first argument is less than the second. False is returned otherwise (e.g., Arg1 < Arg2).

```
lt .Arg1 .Arg2
```

le

Returns a boolean true if the first argument is less than or equal to the second. False is returned otherwise (e.g., Arg1 <= Arg2).

```
le .Arg1 .Arg2
```

Operators are functions

gt

Returns a boolean true if the first argument is greater than the second. False is returned otherwise (e.g., Arg1 > Arg2).

```
gt .Arg1 .Arg2
```

ge

Returns a boolean true if the first argument is greater than or equal to the second. False is returned otherwise (e.g., Arg1 >= Arg2).

```
ge .Arg1 .Arg2
```

String Functions

Helm includes the following string functions:

`abbrev`, `abbrevboth`, `camelcase`, `cat`, `contains`, `hasPrefix`, `hasSuffix`, `indent`, `initials`, `kebabcase`, `lower`, `nindent`, `nospace`, `plural`, `print`, `printf`, `println`, `quote`, `randAlpha`, `randAlphaNum`, `randAscii`, `randNumeric`, `repeat`, `replace`, `shuffle`, `snakecase`, `squote`, `substr`, `swapcase`, `title`, `trim`, `trimAll`, `trimPrefix`, `trimSuffix`, `trunc`, `untitle`, `upper`, `wrap`, and `wrapWith`.

Template syntax - default

To set a simple default value, use default:

```
default "foo" .Bar
```

```
{{ default "minio" .Values.storage }}

//same
{{ .Values.storage | default "minio"}}
```

Flow Control

- `if/else` for creating conditional blocks
- `with` to specify a scope
- `range`, which provides a "for each"-style loop

Template syntax - if/else

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

```
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 }}

## values.yaml

service:
  tls:
    enabled: true
  ...
```

```
ports:
- port: {{ .Values.service.externalPort }}
  targetPort: {{ .Values.service.internalPort }}
  protocol: TCP
{{- if .Values.service.tls }}
  name: "https-{{ .Values.service.name }}"
{{- end }}
```

Serevice scheme and ports

```
type: {{ .Values.service.type}}
ports:
- port: {{ .Values.service.externalPort }}
  targetPort: {{ .Values.service.internalPort }}
  protocol: TCP
{{- if .Values.service.tls }}
  name: "https-{{ .Values.service.name }}"
{{- else }}
  name: "{{ .Values.service.name}}
{{- end }}
```

Template syntax - Modifying scope using with

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

{{ with PIPELINE }}
  # restricted scope
{{ end }}
```

Template syntax - Modifying scope using with

```
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 }}
```

> 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 }}.

Template syntax - Modifying scope using with

This, for example, 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 restricted scope for `.`

```
{{- with .Values.favorite }}
  drink: {{ .drink | default "tea" | quote }}
  food: {{ .food | upper | quote }}
  {{- end }}
  release: {{ .Release.Name }}
```

Template syntax - Modifying scope using with

The following would work as well

```
{{- with .Values.favorite }}
  drink: {{ .drink | default "tea" | quote }}
  food: {{ .food | upper | quote }}
  release: {{ $.Release.Name }}
  {{- end }}
```

> we can use $ for accessing the object Release.Name from the parent scope. $ is mapped to the root scope when template execution begins and it does not change during template execution.

Template syntax - range

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
```

> 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.

Looping with the range action

```
data:
  myvalue: "Hello World"
  {{- with .Values.favorite }}
  drink: {{ .drink | default "tea" | quote }}
  food: {{ .food | upper | quote }}
  {{- end }}
  toppings: |-
    {{- range .Values.pizzaToppings }}
    - {{ . | title | quote }}
    {{- end }}
```

Looping with the range action

```
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 }}
  toppings: |-
    {{- range $.Values.pizzaToppings }}
    - {{ . | title | quote }}
    {{- end }}
  {{- end }}
```

> We can use $ for accessing the list Values.pizzaToppings from the parent scope. $ is mapped to the root scope when template execution begins and it does not change during template execution.

Make a list inside of your template using `tuple`

```
sizes: |-
    {{- range tuple "small" "medium" "large" }}
    - {{ . }}
    {{- end }}
```

The above will produce this:

```
  sizes: |-
    - small
    - medium
    - large
```

Template syntax - variables

```
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 }}
```

> 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: :=

Template syntax - variables

```
{{- 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 }}
```

Declaring and using templates with `define` and `template`

```
{{- define "MY.NAME" }}
  # body of template here
{{- end }}
```

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

```
{{- define "mychart.labels" }}
  labels:
    generator: helm
    date: {{ now | htmlDate }}
{{- end }}
```

Template syntax

```
{{- 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 }}
```

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

Partials

Partials in Helm are reusable templates that can be included in other templates. They help in organizing and reusing code.

Defining a Partial:

```
{{- define "mychart.labels" -}}
labels:
  app: {{ .Values.app.name }}
  version: {{ .Values.app.version }}
{{- end -}}
```

Using a Partial:

```
metadata:
  {{ include "mychart.labels" . | indent 4 }}
```

_ Files

Files that start with an underscore (_) in Helm are not rendered directly but can be used to store partials or helper templates.

Example _helpers.tpl:

```
{{- define "mychart.fullname" -}}
{{- printf "%s-%s" .Release.Name .Chart.Name | trunc 63 | trimSuffix "-" -}}
{{- end -}}
```

Using a Helper from _helpers.tpl:

```
metadata:
  name: {{ include "mychart.fullname" . }}
```

> 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.

Best Practices

`Organize Partials`: Use _helpers.tpl or other _ prefixed files to keep your templates clean and maintainable.

`Naming Conventions`: Use a consistent naming convention for your partials to avoid conflicts and improve readability.

`Indentation`: Use indent and nindent functions to ensure proper YAML formatting when including partials.

> By using partials and _ files, you can create modular, reusable, and maintainable Helm charts.

Helm include vs template

include

Purpose: The include function is used to include the content of another template and return it as a string. This is useful for embedding templates within other templates while maintaining proper formatting.

Usage:

```
{{ include "mychart.labels" . }}
```

Advantages:

`Output Formatting` : The include function allows for better control over the output formatting, especially useful for YAML documents.

`Flexibility` : You can use functions like indent or nindent to adjust the formatting of the included content.

Helm include vs template

template

Purpose: The template function is used to execute a named template and return the result. It is similar to include but does not provide the same level of control over output formatting.

Usage:

```
{{ template "mychart.labels" . }}
```

Disadvantages:

`Less Control`: The template function does not allow for easy manipulation of the output formatting, which can lead to issues in YAML documents.

> 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.

Naming convention

Prefix with Chart Name:

Prefix each defined template with the name of the chart to avoid conflicts and improve readability.

```
{{ define "mychart.labels" }}
labels:
  app: {{ .Values.app.name }}
  version: {{ .Values.app.version }}
{{ end }}
```

Naming convention

Versioning:

Use versioning in template names to manage different versions of templates.

```
{{ define "mychart.v1.labels" }}
labels:
  app: {{ .Values.app.name }}
  version: {{ .Values.app.version }}
{{ end }}

{{ define "mychart.v2.labels" }}
labels:
  app: {{ .Values.app.name }}
  version: {{ .Values.app.version }}
  environment: {{ .Values.app.environment }}
{{ end }}
```

Naming convention

Consistent Naming:

Maintain a consistent naming convention throughout your chart to ensure clarity and maintainability.

```
{{ define "mychart.serviceAccountName" }}
{{ printf "%s-%s" .Release.Name .Chart.Name | trunc 63 | trimSuffix "-" }}
{{ end }}
```

Avoid Special Characters:

Avoid using special characters in template names to prevent issues with rendering and readability.

Use Template Functions

`default` - When the environment value is not provided, it will be defaulted by the template function.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  environment: {{ .Values.environment | default "dev" | quote }}
  region: {{ .Values.region | upper | quote }}
```

Use Template Functions

Specify values that must be set with required:

```
required "A valid foo is required!" .Bar
```

`required` - If the entry is empty, the template rendering will fail with the error `Name is required`.

```
...

metadata:
  name: {{ required "Name is required" .Values.configName }}

...
```

Document Your Charts

README.md

Include a README.md file in your chart directory to provide an overview of the chart, its purpose, and usage instructions.

```
# MyChart

## Overview
MyChart is a Helm chart for deploying a sample application.

## Installation
To install the chart with the release name `my-release`:

helm install my-release ./mychart
...
```

NOTES.txt

Use the `NOTES.txt` file to provide post-installation instructions or important information about the release.

Example:

```
{{- if .Values.service.type == "LoadBalancer" }}
1. Get the application URL by running these commands:
  export SERVICE_IP=$(kubectl get svc --namespace {{ .Release.Namespace }} {{ include "mychart.fullname" . }} \
    -o jsonpath='{.status.loadBalancer.ingress[0].ip}')
  echo http://$SERVICE_IP:{{ .Values.service.port }}
{{- else if .Values.service.type == "ClusterIP" }}
1. Get the application URL by running these commands:
  export POD_NAME=$(kubectl get pods --namespace {{ .Release.Namespace }} \
     -l "app.kubernetes.io/name={{ include "mychart.name" . }},app.kubernetes.io/instance={{ .Release.Name }}" \
     -o jsonpath="{.items[0].metadata.name}")
  echo "Visit http://127.0.0.1:8080 to use your application"
  kubectl port-forward $POD_NAME 8080:80
{{- end }}
```

> The content of the `Note.txt` file can also be templated with functions and values similar to resource templates:

Update Your Deployments When ConfigMaps or Secrets Change

```
kind: Deployment
spec:
  template:
    metadata:
      annotations:
        checksum/config: {{ include (print $.Template.BasePath "/configmap.yaml") . }}
...
```

> Any change in the ConfigMap will calculate a new sha256sum and create new versions of deployment. This ensures the containers in the deployments will restart using the new ConfigMap.

Opt Out of Resource Deletion with Resource Policies

```
kind: Secret
metadata:
  annotations:
    "helm.sh/resource-policy": keep
...
```

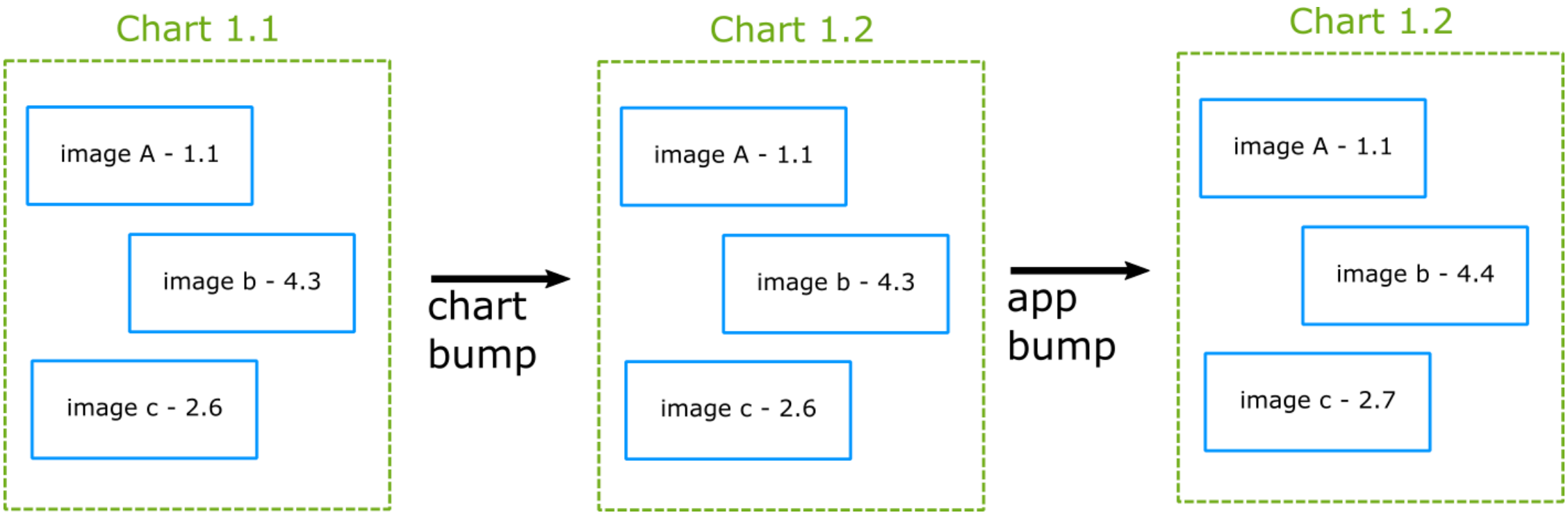
> To keep resources in the cluster even after running Helm uninstall. You need to use the resource-policy annotations as follows

Randomly generate values

```
{{- $rootPasswordValue := (randAlpha 16) | b64enc | quote }}
{{- $secret := (lookup "v1" "Secret" .Release.NameSpace "db-keys") }}
{{- if $secret }}
{{- $rootPasswordValue = index $secret.data "root-password" }}
{{- end -}}
apiVersion: v1
kind: Secret
metadata:
  name: db-keys
  namespace: {{ .Release.Namespace}}
type: Opaque
data:
  root-password: {{ $rootPasswordValue }}
```

> It is recommended to randomly generate values and override those already in the cluster

Chart vs application versioning



разобрать artifacthub.io

как пользоваться комьюнити чартами

деволтные значения посмотреть что в темплейтах

версия приложения версия образа

заметки по установке и обновлению версий

лучшие практики храним у себя обновляем репозитоий автоматический по мере необходимости

Helm Chart Hooks Use Cases

- Loading of secrets to access a repository to pull an image before the main service is deployed
- To perform DB migrations before updating the service
- Cleaning up external resources after the service is deleted
- Checking for the prerequisites of a service before the service is deployed

Hook example

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  annotations:
    # This is what defines this resource as a hook. Without this line, the
    # job is considered part of the release.
    "helm.sh/hook": post-install, post-upgrade
    "helm.sh/hook-weight": "-5"
```

Keep CD Pipilines Idempotent

There are two essential rules to follow:

- Always use the `helm upgrade --install` command. It installs the charts if they are not already installed. If they are already installed, it upgrades them.
- Use `--atomic` flag to rollback changes in the event of a failed operation during helm upgrade. This ensures the Helm releases are not stuck in the failed state.

Valid chart names

Valid:

- nginx
- wordpress
- wordpress-on-nginx

Invalid names:

- Nginx
- Wordpress
- wordpressOnNginx
- wordpress_on_nginx

The values.yaml file

Use "camelcase": the first word starts with a lowercase letter, but the next ones all start with a capital letter.

Examples of valid names:

```
replicaCount: 3
wordpressUsername: user
wordpressSkipInstall: false
```

A values file is passed into helm install or helm upgrade with the `-f flag` (`helm install -f myvals.yaml ./mychart`)

Individual parameters are passed with `--set` (such as `helm install --set foo=bar ./mychart`)

The .helmignore file

```
# comment

# Match any file or path named .helmignore
.helmignore

# Match any file or path named .git
.git

# Match any text file
*.txt

# Match only directories named mydir
mydir/

# Match only text files in the top-level directory
/*.txt

# Match only the file foo.txt in the top-level directory
/foo.txt

# Match any file named ab.txt, ac.txt, or ad.txt
a[b-d].txt

# Match any file under subdir matching temp*
*/temp*

*/*/temp*
temp?
```

Flat vs. Nested Values

We saw in our exercises that we can have nested values, like this:

```
image:
  repository:
  pullPolicy: IfNotPresent
  tag: "1.16.0"
```

This provides some logical grouping, in this case, image is the parent variable that has three children values. But this can be rewritten to a flat format, like this:

```
imageRepository:
imagePullPolicy: IfNotPresent
imageTag: "1.16.0"
```

Quote Strings

enabled: false and **enabled: “false”** will assign different types of values to the enabled variable.

Always wrap your string values between `“”` quote signs.

Практика

LAB: Create you first helm chart

https://github.com/AliaksandrTsimokhau/devops_pro_dtu/blob/main/helm_chart_lab.md

usefull materials: helm_cheatsheet.md. slides for the lesson