KUB 30: Углубленная настройка Helm

Описание:

Helm также предоставляет hooks-механизм, позволяющий выполнять правильные действия при установке/удалении/обновлении или откате чарта. Например, указанные hooks можно использовать для решения следующих задач:

- применять в кластере такие ресурсы, как ConfigMap/Secrets перед тем, как будут применены ресурсы, в которых используются указанные ConfigMap/Secret;
- выполнить job для резервирования данных БД, миграций или тестового наполнения БД данными перед обновлением чарта;
- запустить какое-либо специфическое задание для graceful удаления нагрузки из кластера.

Доступные Hooks:

- pre-install — выполняется перед рендерингом чарта при установке;
- post-install — выполняется после рендеринга и установки всех темплейтов;
- pre-delete — выполняется перед процедурой удаления чарта, до того как непосредственно происходит удаление ресурсов;
- post-delete — выполняется после удаления чарта;
- pre-upgrade — выполняется после запроса на обновление чарта, после рендеринга конфигов, но до того, как обновленные ресурсы будут применены;
- post-upgrade — выполняется после обновления ресурсов чарта;
- pre-rollback — выполняется после получения запроса на rollback, после рендеринга конфигов, но до изменения ресурсов;
- post-rollback — выполняется после изменения ресурсов при rollback;
- test — выполняется при вызове подкоманды test для helm.

В Helm существует механизм для подключения иных чартов как зависимостей по аналогии с тем, как пакеты в дистрибутивах Linux могут требовать установки иных пакетов.
Существует 2 метода определения зависимостей:

1. Через chart.yaml — актуально для версии Helm 3.
2. Через requirements.yaml — актуально для версий Helm 2 и 3.

Разница в методах определения появилась с приходом Helm версии 3. То, где должны храниться зависимости;определяется тем, какая apiVersion выставлена в chart.yaml в Helm Chart. Первая версия актуальна со времен Helm 2 и хранила зависимости именно в requirements.yaml. Helm 3 поддерживает работу с apiVersion=1, поэтому логика работы должна быть одинакова с любой версией Helm.
Давайте посмотрим, как выглядит определение зависимостей:

```
dependencies:
  - name: kube-state-metrics
    version: 2.4.*
```

```
    repository:
https://kubernetes-charts.storage.googleapis.com/
    alias: kubestatemetrics
    condition: kubeStateMetrics.enabled
    tags:
    - monitoring
```

Как видно, зависимостей может быть несколько, так как это массив. сейчас пройдемся по полям зависимости:

1. name — имя зависимости;
2. version — версия чарта, который должен быть получен;
3. repository — адрес репозитория, из которого нужно запросить архив с чартом;
4. condition — указание boolean поля в Values, которое должно быть в значении true, для того чтобы чарт был установлен. Может содержать несколько значений, разделенных запятой. Если хотя бы одно из значений отсутствует или равняется false, то зависимость не установится;
5. tags — схоже с condition, но требует наличия boolean значения с именем тега в блоке tags в values.

Полезное замечание — если по condition или по tags чарт должен быть установлен, то он будет установлен, даже в случае разночтения значений (скажем, tag — false, но связанный condition — true).
При помощи Values в чарте можно устанавливать значения и для зависимых чартов. Так, предположим, у нас есть чарт с именем sub1, у которого следующий values.yaml:

```
sub1_password: qwerty
subblock:
  enabled: true
```

Для того чтобы установить значения для subchart в values.yaml нашего чарта, требуется описать его следующим образом:

```
db_password: ytrewq

sub1:
  sub1_password: ytrewq
  subblock:
    enabled: false
```

Раз уж мы начали разбираться зачем нужны зависимости, стоит рассказать, для чего в tempates/_helpers.tpl у нас есть шаблоны для CHART.name, CHART.fullname и так далее. Они нужны для того, чтобы в случае множественной установки одного и того же чарта с разными значениями у нас создавались не захардкорженные значения, а значения, связанные именно с релизом, а не чартом (скажем, имена Deployment). В противном же

случае установка той же базы данных приводила бы к созданию манифестов с одинаковыми именами, из-за чего она бы завершалась ошибкой.
Хорошо, зависимости мы описали, но как ими пользоваться? Есть явный и неявный путь.

- Явный — используя команды, связанные с helm dependency, — они позволяют создать requirements.lock файлы (аналогично package.lock в NodeJS приложениях, что позволяет жестко закрепить версии зависимостей, обновить зависимости и вывести список зависимостей).
- Неявный — просто установив чарт. С этим все несложно — при установке Helm сам скачает и положит зависимости в директорию charts, поскольку без них не сможет запустить приложение с зависимостями.

После деплоя чарта helm собирает сгенерированные манифесты и сохраняет их в storage backend. По умолчанию, вся информация хранится в виде ConfigMap — для каждой версии релиза создается своя.
Далее при каждом апгрейде релиза helm сравнивает информацию сгенерированных манифестов с теми, что хранятся в storage backend (последней DEPLOYED версии релиза), и применяет разницу.
Указанную информацию можно получить при помощи следующей команды:

```
kubectl get cm
```

Историю релиза можно получить с помощью команды —

```
helm history chart_name
```

Helm rollback предоставляет возможность откатиться на конкретную версию релиза из истории.
Полное удаление релиза, включая всю метаинформацию о нем, из кластера возможно при помощи ключа --purge для команды helm delete.
Helm является очень вариативным инструментом с огромным множеством возможностей. В этом обзоре будет приведено лишь небольшое количество хитростей, которые можно использовать в работе с ним, да и список, приведенный в официальной документации, не полон. Однако, если вы хотите лучше понять этот инструмент и делать множество вещей наиболее элегантным, с технической точки зрения, образом, то эта информация послужит хорошим фундаментом.

hash templating

Достаточно часто передаваемые изменения в чарте при обновлении затрагивают такие элементы, как ConfigMap/Secrets. При этом сами Deployments, которые используют конфигурации из ConfigMap/Secret, остаются неизменными. Это может привести к тому, что при обновлении ресурсов поды продолжат работать на старых конфигах, так как они не были перезапущены.
Чтобы избежать этой проблемы, вы можете использовать sha256sum для определения изменения ресурсов и при их изменении обновлять связанные ресурсы. Подробнее можно почитать тут.

Golang magic

При использовании шаблонизатора Golang вы часто будете сталкиваться с необходимостью более глубокого взаимодействия с инструментом, чем простая подстановка данных из values. Например, можно использовать дополнительную логику с проверкой наличия значения:

```
value: {{ required "A valid .Values.who entry required!"
.Values.who }}
```

Данный подход позволяет еще на этапе рендеринга конфигурационных файлов выявить проблему в чарте.
Indent-функция определяет количество пробелов от начала вставки, эта опция позволяет легко соблюдать структуру yaml-формата:

```
{{ include "toYaml" $value | indent 2 }}
```

Генерация таких элементов, как ImagePullSecret, может быть произведена несколькими путями. Вы можете просто взять и загнать base64 от конфига с авторизационными данными в соответствующий секрет, а можете сгенерировать то же самое, используя tpl:

```
Values.yml: |
  imageCredentials:
    registry: quay.io
    username: someone
    password: sillyness

tpl: |
  {{- define "imagePullSecret" }}
  {{- printf "{\"auths\": {\"%s\": {\"auth\": \"%s\"}}}"
.Values.imageCredentials.registry (printf "%s:%s"
.Values.imageCredentials.username
.Values.imageCredentials.password | b64enc) | b64enc }}
  {{- end }}

secret.yaml: |
  data:
    .dockerconfigjson: {{ template "imagePullSecret" . }}
```

И множество иных хитростей...

Helm install/upgrade

В процессе организации CI/CD вам необходимо иметь универсальную команду, которая будет устанавливать чарт, если его нет, обновлять, если он уже существует, еще и делать это максимально надежно. Такой командой является
`helm upgrade chart_name --install ./helm_chart/`

Начиная с helm 3, появился полезный флаг --create-namespace, - который позволяет создать соответсвующий namespace (используется совместно с --namespace name). В 2.x версиях namespace должен быть создан заранее.

Debug

В процессе составления чарта вам может потребоваться наличие дебаг-инструмента. Разумеется, есть Helm lint, но что же еще?
Командой Helm template можно отрендерить ваши конфигурационные файлы, имитируя установку. В выводе, соответственно, будут все сгенерированные манифесты. Для дебага yaml-формата можно использовать утилиту yq , работая с выводом команды helm template. Для более детального вывода, вы можете использовать флаги --dry-run и --debug

Полезные ссылки:

- Helm Docs: Tips and Tricks
- Charts (official docs)
- Dependencies (official docs)
- helm-dependency example (github)
- Helm Docs: hooks

Задание:

1. Задеплойте чарт из предыдущего задания с именем релиза local-chart в namespace helm.
2. Измените количество реплик до 4 в чарте.
3. Задеплойте изменения с помошью helm в local-chart в namespace helm.
4. Откатите релиз local-chart до первой ревизии с помощью helm.
5. Подождите, пока все реплики будут уничтожены (в живых должен остаться только 1 под!). Только после этого отправьте задание на проверку.