Helm - это менеджер пакетов для Kubernetes. Этот инструмент позволяет нам обернуть Kubernetes приложения в удобные пакеты, называемые чартами, которые можно легко развертывать, обновлять и управлять ими в любой момент времени.

Чарты – это пакеты, которые могут включать в себя все для запуска приложения в Kubernetes, от deployments до services. Все это дает возможность работать с приложениями как с единой сущностью, а не как с набором отдельных ресурсов, которые еще и в ручную нужно настраивать...

Так же Helm **упрощает управление зависимостями** между приложениями, позволяет легко параметризировать настройки приложений через файлы values.yaml и дает возможность повторного использования чартов с помощью шаблонизации.

К тому же можно с легкостью откатиться к предыдущей версии нашего приложения.

Структура Helm Chart

Структура Helm Chart представляет собой директорию, которая содержит комплекс механизмов для управления приложениями, упаковывая их в чарты, которые можно легко инсталлировать, обновлять и управлять.

Первый и самый главный файл **- Chart.yaml**, он содержит метаданные: имя чарта, версию, описание, информацию о зависимостях и т.д. Этот файл обязателен и помогает идентифицировать чарт:

```yaml
apiVersion: v2
name: mychart
version: 1.0.0
description: "helm chart"
keywords:
  - mykeyword
home: http://example.com/
sources:
  - https://github.com/example/mychart
dependencies:
  - name: nginx
    version: "1.14.0"
    repository: "https://charts.helm.sh/stable"
```

Values.yaml определяет конфигурационные параметры, которые можно переопределить во время установки или обновления чарта. Эти значения используются в шаблонах чарта для динамической генерации Kubernetes манифестов:

```
replicaCount: 2
image:
  repository: nginx
  tag: stable
  pullPolicy: IfNotPresent
service:
  type: NodePort
  port: 80
```

Директория **templates/** содержит шаблоны манифестов Kubernetes, которые используют как статические, так и динамические данные из values.yaml и Chart.yaml для генерации конечных манифестов. Helm обрабатывает каждый файл в этой директории, применяя шаблонизацию для создания ресурсов Kubernetes. Например, templates/service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: {{ include "mychart.fullname" . }}
  labels:
    {{- include "mychart.labels" . | nindent 4 }}
spec:
  type: {{ .Values.service.type }}
  ports:
    - port: {{ .Values.service.port }}
      targetPort: 80
  selector:
    app.kubernetes.io/name: {{ include "mychart.name" . }}
    app.kubernetes.io/instance: {{ .Release.Name }}
```

templates/deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "mychart.fullname" . }}
  labels:
    {{- include "mychart.labels" . | nindent 4 }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app.kubernetes.io/name: {{ include "mychart.name" . }}
      app.kubernetes.io/instance: {{ .Release.Name }}
  template:
```

```
    metadata:
      labels:
        app.kubernetes.io/name: {{ include "mychart.name" . }}
        app.kubernetes.io/instance: {{ .Release.Name }}
    spec:
      containers:
        - name: nginx
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
          ports:
            - containerPort: 80
```

templates/_helpers.tpl:

```
{{/*
Expand the name of the chart.
*/}}
{{- define "mychart.name" -}}
{{- default .Chart.Name .Values.nameOverride | trunc 63 | trimSuffix "-" -}}
{{- end -}}
```

Charts/ - это директория для подчартов, т.е., зависимостей вашего чарта. Если приложение требует другие чарты для своей работы, они размещаются здесь

.helmignore - файл, аналогичный .gitignore, позволяющий исключить файлы и директории из пакета чарта, что уменьшает его размер и предотвращает включение в чарт лишних данных и т.д:

```
# игнорить все файлы .git и .svn
.git
.svn

# игнорить все файлы .md, кроме README.md
*.md
!README.md

# игнорить специфичные файлы и директории
tmp/
temp/
secrets.yaml
```

Файл **NOTES.txt** в директории templates предоставляет пользователю информацию после установки чарта, например, как подключиться к приложению или следующие шаги после установки.:

```
1. Get the application URL by running these commands:
  {{- if .Values.service.type == "NodePort" }}
  export NODE_PORT=$(kubectl get --namespace {{ .Release.Namespace }} -o jsonpath="{.spec.po
  export NODE_IP=$(kubectl get nodes --namespace {{ .Release.Namespace }} -o jsonpath="{.ite
  echo http://$NODE_IP:$NODE_PORT
  {{- end }}
```

Этот файл также поддерживает шаблонизацию.

Управление хранилищами

Helm использует хранилище для хранения информации о релизах, и по умолчанию в Helm 3 это хранилище реализовано с использованием `ConfigMaps`. Однако существуют альтернативные хранилища, такие как Secrets или хранение в базе данных SQL, которые в разы удобней.

Secrets - это базовый выбор для хранения чувствительной информации: пароли или ключи, связанные с Helm релизами.

Настройка Secrets:

```yaml
# values.yaml для чарта

secrets:
  enabled: true
  backend: secrets
```

```yaml
# файл templates/secrets.yaml

apiVersion: v1
kind: Secret
metadata:
  name: {{ .Release.Name }}-secrets
type: Opaque
data:
  # можно указать данные, к примеру пароль
```

Для других сценариев, когда требуется более вместительное и функциональное хранилище, можно использовать SQL базу данных. Например, можно использовать PostgreSQL:

```yaml
# values.yaml для чарта
```

```
sql:
  enabled: true
  backend: postgresql
  postgresqlHost: my-postgresql-host
  postgresqlDatabase: helm_releases
  postgresqlUsername: helm
  postgresqlPassword: qazwsxedcdanormparol
```

```
CREATE TABLE IF NOT EXISTS releases (
    release_name VARCHAR(255) PRIMARY KEY,
    chart_name VARCHAR(255),
    chart_version VARCHAR(255),
    release_version INT,
    status VARCHAR(255),
    created_at TIMESTAMP,
    updated_at TIMESTAMP
);
```

Post-Renderers

Post-Renderers дает возможность модифицировать манифесты Kubernetes после их генерации шаблонизатором Helm, но до того, как они будут применены в Kubernetes. Это дает возможности для кастомизации, позволяя интегрировать дополнительные инструменты и процессы в workflow развертывания (например, kustomize)

Допустим, есть Helm чарт, который разворачивает приложение в Kubernetes, и я хочу модифицировать созданные манифесты перед развертыванием, используя Kustomize для добавления специфичных настроек безопасности или для изменения конфигурации ресурсов.

Сначала создается директория, например `kustomize`, внутри проекта и в нее добавляем файл `kustomization.yaml` со всеми необходимыми настройками. Например, можно добавить новые метки или аннотации к Deployment:

```
# kustomize/kustomization.yaml
commonLabels:
  environment: production

patchesStrategicMerge:
  - patch.yaml
```

К тому же **обязательно создаем** `patch.yaml` **в той же директории** для изменения спецификации Deployment:

```yaml
# kustomize/patch.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-application
spec:
  template:
    spec:
      containers:
      - name: my-application
        resources:
          limits:
            cpu: "500m"
            memory: "128Mi"
```

Создаем скрипт, который будет использоваться как post-renderer. Этот скрипт должен читать манифесты из STDIN, применять к ним Kustomize и выводить результат в STDOUT:

```bash
#!/bin/bash

# сейвим входные данные во временный файл
INPUT=$(mktemp)
cat > $INPUT

# юзаем Kustomize
kustomize build kustomize | cat

# дел временных файлов
rm $INPUT
```

Делаем скрипт исполняемым:

```bash
chmod +x post-render.sh
```

Теперь можно юзать этот скрипт в качестве post-renderer при установке или обновлении чарта:

```bash
helm install my-chart ./mychart --post-renderer ./post-render.sh
```

Или при обновлении:

```
helm upgrade my-chart ./mychart --post-renderer ./post-render.sh
```

Helm Dependencies

Часто приложение или сервис может зависеть от других чартов. Например, приложение может требовать бдшки Redis или сервера сообщений RabbitMQ. Вместо того, чтобы каждый раз вручную устанавливать и настраивать эти зависимости, Helm позволяет автоматически управлять ими через файл `Chart.yaml` чарта.

Зависимости определяются в файле `Chart.yaml` или `requirements.yaml` в Helm чарте. Там указываются имя чарта зависимости, версию, и репозиторий, откуда Helm может загрузить чарт:

```
apiVersion: v2
name: my-application
version: 1.0.0
dependencies:
  - name: redis
    version: "^6.0.1"
    repository: "https://charts.bitnami.com/bitnami"
  - name: rabbitmq
    version: "8.0.2"
    repository: "https://charts.bitnami.com/bitnami"
```

После определения зависимости в `Chart.yaml`, можно использовать команды Helm для управления этими зависимостями:

- **helm dependency list** показывает список зависимостей и их состояние.
- **helm dependency update** скачивает и размещает зависимости в директорию `charts/` чарта.
- **helm dependency build** обновляет зависимости и регенерирует файл `charts.lock`, не скачивая пакеты заново.
- **helm dependency update** делает то же самое, что и `build`, но также скачивает все необходимые чарты.

Допустим, мы воркаем над приложением, которое требует Redis. Вместо того, чтобы каждый раз вручную развертывать Redis, можно просто определить его как зависимость в чарте:

1. определяем зависимость в `Chart.yaml`.
2. запускаем `helm dependency update` для загрузки и упаковки чарта Redis в чарт.
3. когда вы запускаем чарт с помощью `helm install`, Helm также установит Redis, используя определенные настройки.

Helm SDK

В golang есть Helm SDK, который позволяет программно управлять Helm чартами, репозиториями и релизами в Kubernetes.

В основе работы с Helm SDK лежит конфигурация клиента `action.Configuration` которая включает в себя настройки для взаимодействия с Kubernetes кластером и Tiller или непосредственно с Kubernetes API. `action.Configuration` используется для инициализации различных клиентов, предназначенных для выполнения операций Helm, таких как установка `action.Install`, обновление `action.Upgrade`, удаление `action.Uninstall` и многие другие.

SDK дает функции для работы с чартами, включая их поиск, установку, обновление и удаление. Можно юзать эти функции для автоматизации деплоймента приложений, а также для создания сложных пайплайнов, которые автоматически обновляют чарты в зависимости от изменений в исходном коде или конфигурации.

С помощью SDK можно управлять релизами Helm, включая получение информации о текущих релизах, их истории, а также откат к предыдущим версиям. Это основная возможность в хелм для реализации стратегий Blue/Green или Canary деплойментов

Первый шаг - инициализация клиента Helm:

```
package main

import (
    "helm.sh/helm/v3/pkg/action"
    "helm.sh/helm/v3/pkg/cli"
    "k8s.io/client-go/util/homedir"
    "path/filepath"
)

func main() {
    settings := cli.New()
    actionConfig := new(action.Configuration)

    kubeconfig := filepath.Join(homedir.HomeDir(), ".kube", "config")
    if err := actionConfig.Init(settings.RESTClientGetter(), settings.Namespace(), os.Getenv
        fmt.Printf(format, v...)
    }); err != nil {
        log.Fatalf("Failed to initialize Helm client: %v", err)
    }
}
```

Для установки чарта с помощью Go SDK, можно использовать клиент `action.Install` и конфигурировать его параметры, аналогично тому, как бы делали это с использованием CLI

команды `helm install`:

```
install := action.NewInstall(actionConfig)
install.ReleaseName = "my-release"
install.Namespace = "default"
chartPath, err := install.LocateChart("mychart", settings)
if err != nil {
    log.Fatalf("Failed to locate chart: %v", err)
}

_, err = install.Run(chartPath, nil) // Второй параметр может содержать значения для переопр
if err != nil {
    log.Fatalf("Failed to install chart: %v", err)
}
```

Обновление чарта работает аналогично установке, но использует клиент `action.Upgrade`:

```
upgrade := action.NewUpgrade(actionConfig)
upgrade.Namespace = "default"
if _, err := upgrade.Run("my-release", chartPath, nil); err != nil {
    log.Fatalf("Failed to upgrade chart: %v", err)
}
```

Заключение

Helm Charts позволяют упростить управление приложениями и автоматизировать многие процессы.