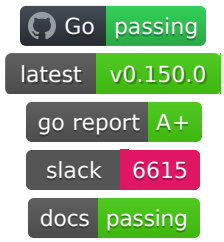
Helmfile



Deploy Kubernetes Helm Charts

Status

March 2022 Update - The helmfile project has been moved to helmfile/helmfile from the former home `roboll/helmfile`. Please see roboll/helmfile#1824 for more information.

Even though Helmfile is used in production environments across multiple organizations, it is still in its early stage of development, hence versioned 0.x.

Helmfile complies to Semantic Versioning 2.0.0 in which v0.x means that there could be backward-incompatible changes for every release.

Note that we will try our best to document any backward incompatibility. And in reality, helmfile had no breaking change for a year or so.

About

Helmfile is a declarative spec for deploying helm charts. It lets you...

- Keep a directory of chart value files and maintain changes in version control.
- Apply CI/CD to configuration changes.
- Periodically sync to avoid skew in environments.

To avoid upgrades for each iteration of `helm`, the `helmfile` executable delegates to `helm` - as a result, `helm` must be installed.

Highlights

Declarative: Write, version-control, apply the desired state file for visibility and reproducibility.

Modules: Modularize common patterns of your infrastructure, distribute it via Git, S3, etc. to be reused across the entire company (See #648)

Versatility: Manage your cluster consisting of charts, kustomizations, and directories of Kubernetes resources, turning everything to Helm releases (See #673)

Patch: JSON/Strategic-Merge Patch Kubernetes resources before `helm-install`ing, without forking upstream charts (See #673)

Installation

- download one of releases
- run as a container
- Archlinux: install via `pacman -S helmfile`
- openSUSE: install via `zypper in helmfile` assuming you are on Tumbleweed; if you are on Leap you must add the kubic repo for your distribution version once before that command, e.g. `zypper ar https://download.opensuse.org/repositories/devel:/kubic/openSUSE_Leap_\$releasever kubic`
- Windows (using scoop): `scoop install helmfile`
- macOS (using homebrew): `brew install helmfile`

Running as a container

The Helmfile Docker images are available in GHCR. There is no `latest` tag, since the `0.x` versions can contain breaking changes, so make sure you pick the right tag. Example using `helmfile 0.145.2`:

```
# helm 2
$ docker run --rm --net=host -v "${HOME}/.kube:/root/.kube" -v "${HOME}/.helm:/root/.helm

# helm 3
$ docker run --rm --net=host -v "${HOME}/.kube:/root/.kube" -v "${HOME}/.config/helm:/roo
```

You can also use shims to make calling the binaries easier:

```
# helm 2
$ printf '%s\n' '#!/bin/sh' 'docker run --rm --net=host -v "${HOME}/.kube:/root/.kube" -v
    tee helmfile
$ chmod +x helmfile
$ ./helmfile sync

# helm 3
$ printf '%s\n' '#!/bin/sh' 'docker run --rm --net=host -v "${HOME}/.kube:/root/.kube" -v
    tee helmfile
$ chmod +x helmfile
$ ./helmfile sync
```

Getting Started

Let's start with a simple `helmfile` and gradually improve it to fit your use-case!

Suppose the `helmfile.yaml` representing the desired state of your helm releases looks like:

```yaml
repositories:
 - name: prometheus-community
   url: https://prometheus-community.github.io/helm-charts

releases:
- name: prom-norbac-ubuntu
  namespace: prometheus
  chart: prometheus-community/prometheus
  set:
  - name: rbac.create
    value: false
```

Sync your Kubernetes cluster state to the desired one by running:

```
helmfile apply
```

Congratulations! You now have your first Prometheus deployment running inside your cluster.

Iterate on the `helmfile.yaml` by referencing:

- Configuration
- CLI reference.
- Helmfile Best Practices Guide

Configuration

CAUTION: This documentation is for the development version of Helmfile. If you are looking for the documentation for any of releases, please switch to the corresponding release tag like v0.143.4.

The default name for a helmfile is `helmfile.yaml`:

v: latest

```yaml
# Chart repositories used from within this state file
#
# Use `helm-s3` and `helm-git` and whatever Helm Downloader plugins
# to use repositories other than the official repository or one backend by chartmuseum.
repositories:
# To use official "stable" charts a.k.a https://github.com/helm/charts/tree/master/stable
- name: stable
  url: https://charts.helm.sh/stable
# To use official "incubator" charts a.k.a https://github.com/helm/charts/tree/master/inc
- name: incubator
  url: https://charts.helm.sh/incubator
# helm-git powered repository: You can treat any Git repository as a charts repository
- name: polaris
  url: git+https://github.com/reactiveops/polaris@deploy/helm?ref=master
# Advanced configuration: You can setup basic or tls auth and optionally enable helm OCI
- name: roboll
  url: roboll.io/charts
  certFile: optional_client_cert
  keyFile: optional_client_key
  # username is retrieve from the environment with the format <registryNameUpperCase>_USE
  username: optional_username
  # username is retrieve from the environment with the format <registryNameUpperCase>_PAS
  password: optional_password
  oci: true
  passCredentials: true
# Advanced configuration: You can use a ca bundle to use an https repo
# with a self-signed certificate
- name: insecure
   url: https://charts.my-insecure-domain.com
   caFile: optional_ca_crt
# Advanced configuration: You can skip the verification of TLS for an https repo
- name: skipTLS
  url: https://ss.my-insecure-domain.com
  skipTLSVerify: true

# context: kube-context # this directive is deprecated, please consider using helmDefault

# Path to alternative helm binary (--helm-binary)
helmBinary: path/to/helm3


# Path to alternative lock file. The default is <state file name>.lock, i.e for helmfile.
lockFilePath: path/to/lock.file

# Default values to set for args along with dedicated keys that can be set by contributor
# In other words, unset values results in no flags passed to helm.
# See the helm usage (helm SUBCOMMAND -h) for more info on default values when those flag
helmDefaults:
  kubeContext: kube-context          #dedicated default key for kube-context (--kube-cont
  cleanupOnFail: false               #dedicated default key for helm flag --cleanup-on-fa
  # additional and global args passed to helm (default "")
  args:
    - "--set k=v"
  # verify the chart before upgrading (only works with packaged charts not directories) (
  verify: true
  # wait for k8s resources via --wait. (default false)
  wait: true
  # if set and --wait enabled, will wait until all Jobs have been completed before markin
  waitForJobs: true
  # time in seconds to wait for any individual Kubernetes operation (like Jobs for hooks,
  timeout: 600
  # performs pods restart for the resource if applicable (default false)
  recreatePods: true
  # forces resource update through delete/recreate if needed (default false)
  force: false
  # enable TLS for request to Tiller (default false)
  tls: true
  # path to TLS CA certificate file (default "$HELM_HOME/ca.pem")
  tlsCACert: "path/to/ca.pem"
  # path to TLS certificate file (default "$HELM_HOME/cert.pem")
  tlsCert: "path/to/cert.pem"
  # path to TLS key file (default "$HELM_HOME/key.pem")
  tlsKey: "path/to/key.pem"
  # limit the maximum number of revisions saved per release. Use 0 for no limit. (default
  historyMax: 10
  # when using helm 3.2+, automatically create release namespaces if they do not exist (d
```

```yaml
  createNamespace: true
  # if used with charts museum allows to pull unstable charts for deployment, for example
  devel: true
  # When set to `true`, skips running `helm dep up` and `helm dep build` on this release'
  # Useful when the chart is broken, like seen in https://github.com/roboll/helmfile/issu
  skipDeps: false
  # If set to true, reuses the last release's values and merges them with ones provided i
  # This attribute, can be overriden in CLI with --reset/reuse-values flag of apply/sync/
  reuseValues: false
  # propagate `--post-renderer` to helmv3 template and helm install
  postRenderer: "path/to/postRenderer"

# these labels will be applied to all releases in a Helmfile. Useful in templating if you
commonLabels:
  hello: world

# The desired states of Helm releases.
#
# Helmfile runs various helm commands to converge the current state in the live cluster t
releases:
  # Published chart example
  - name: vault                           # name of this release
    namespace: vault                      # target namespace
    createNamespace: true                 # helm 3.2+ automatically create release names
    labels:                               # Arbitrary key value pairs for filtering rele
      foo: bar
    chart: roboll/vault-secret-manager    # the chart being installed to create this rel
    version: ~1.24.1                      # the semver of the chart. range constraint is
    condition: vault.enabled              # The values lookup key for filtering releases
    missingFileHandler: Warn # set to either "Error" or "Warn". "Error" instructs helmfil
    # Values files used for rendering the chart
    values:
      # Value files passed via --values
      - vault.yaml
      # Inline values, passed via a temporary values file and --values, so that it doesn'
      - address: https://vault.example.com
      # Go template available in inline values and values files.
      - image:
          # The end result is more or less YAML. So do `quote` to prevent number-like str
          # See https://github.com/roboll/helmfile/issues/608
          tag: {{ requiredEnv "IMAGE_TAG" | quote }}
          # Otherwise:
          #   tag: "{{ requiredEnv "IMAGE_TAG" }}"
          #   tag: !!string {{ requiredEnv "IMAGE_TAG" }}
        db:
          username: {{ requiredEnv "DB_USERNAME" }}
          # value taken from environment variable. Quotes are necessary. Will throw an er
          password: {{ requiredEnv "DB_PASSWORD" }}
        proxy:
          # Interpolate environment variable with a fixed string
          domain: {{ requiredEnv "PLATFORM_ID" }}.my-domain.com
          scheme: {{ env "SCHEME" | default "https" }}
    # Use `values` whenever possible!
    # `set` translates to helm's `--set key=val`, that is known to suffer from type issue
    set:
    # single value loaded from a local file, translates to --set-file foo.config=path/to/
    - name: foo.config
      file: path/to/file
    # set a single array value in an array, translates to --set bar[0]={1,2}
    - name: bar[0]
      values:
      - 1
      - 2
    # set a templated value
    - name: namespace
      value: {{ .Namespace }}
    # will attempt to decrypt it using helm-secrets plugin
    secrets:
      - vault_secret.yaml
    # Override helmDefaults options for verify, wait, waitForJobs, timeout, recreatePods
    verify: true
    wait: true
    waitForJobs: true
    timeout: 60
    recreatePods: true
    force: false
    # set `false` to uninstall this release on sync.  (default true)
```

```yaml
    installed: true
    # restores previous state in case of failed release (default false)
    atomic: true
    # when true, cleans up any new resources created during a failed release (default fal
    cleanupOnFail: false
    # enable TLS for request to Tiller (default false)
    tls: true
    # path to TLS CA certificate file (default "$HELM_HOME/ca.pem")
    tlsCACert: "path/to/ca.pem"
    # path to TLS certificate file (default "$HELM_HOME/cert.pem")
    tlsCert: "path/to/cert.pem"
    # path to TLS key file (default "$HELM_HOME/key.pem")
    tlsKey: "path/to/key.pem"
    # --kube-context to be passed to helm commands
    # CAUTION: this doesn't work as expected for `tilerless: true`.
    # See https://github.com/roboll/helmfile/issues/642
    # (default "", which means the standard kubeconfig, either ~/kubeconfig or the file p
    kubeContext: kube-context
    # passes --disable-validation to helm 3 diff plugin, this requires diff plugin >= 3.1
    # It may be helpful to deploy charts with helm api v1 CRDS
    # https://github.com/roboll/helmfile/pull/1373
    disableValidation: false
    # passes --disable-validation to helm 3 diff plugin, this requires diff plugin >= 3.1
    # It is useful when any release contains custom resources for CRDs that is not yet in
    # https://github.com/roboll/helmfile/pull/1618
    disableValidationOnInstall: false
    # passes --disable-openapi-validation to helm 3 diff plugin, this requires diff plugi
    # It may be helpful to deploy charts with helm api v1 CRDS
    # https://github.com/roboll/helmfile/pull/1373
    disableOpenAPIValidation: false
    # limit the maximum number of revisions saved per release. Use 0 for no limit (defaul
    historyMax: 10
    # When set to `true`, skips running `helm dep up` and `helm dep build` on this releas
    # Useful when the chart is broken, like seen in https://github.com/roboll/helmfile/is
    skipDeps: false
    # propagate `--post-renderer` to helmv3 template and helm install
    postRenderer: "path/to/postRenderer"

  # Local chart example
  - name: grafana                           # name of this release
    namespace: another                      # target namespace
    chart: ../my-charts/grafana             # the chart being installed to create this r
    values:
    - "../../my-values/grafana/values.yaml"             # Values file (relative path to m
    - ./values/{{ requiredEnv "PLATFORM_ENV" }}/config.yaml # Values file taken from path
    wait: true

#
# Advanced Configuration: Nested States
#
helmfiles:
- # Path to the helmfile state file being processed BEFORE releases in this state file
  path: path/to/subhelmfile.yaml
  # Label selector used for filtering releases in the nested state.
  # For example, `name=prometheus` in this context is equivalent to processing the nested
  #   helmfile -f path/to/subhelmfile.yaml -l name=prometheus sync
  selectors:
  - name=prometheus
  # Override state values
  values:
  # Values files merged into the nested state's values
  - additional.values.yaml
  # One important aspect of using values here is that they first need to be defined in th
  # of the origin helmfile, so in this example key1 needs to be in the values or environm
  # Inline state values merged into the nested state's values
  - key1: val1
- # All the nested state files under `helmfiles:` is processed in the order of definition
  # So it can be used for preparation for your main `releases`. An example would be creat
  path: path/to/mycrd.helmfile.yaml
- # Terraform-module-like URL for importing a remote directory and use a file in it as a
  # The nested-state file is locally checked-out along with the remote directory containi
  # Therefore all the local paths in the file are resolved relative to the file
  path: git::https://github.com/cloudposse/helmfiles.git@releases/kiam.yaml?ref=(
# If set to "Error", return an error when a subhelmfile points to a
# non-existent path. The default behavior is to print a warning and continue.
missingFileHandler: Error
```

```yaml
#
# Advanced Configuration: Environments
#

# The list of environments managed by helmfile.
#
# The default is `environments: {"default": {}}` which implies:
#
# - `{{ .Environment.Name }}` evaluates to "default"
# - `{{ .Values }}` being empty
environments:
  # The "default" environment is available and used when `helmfile` is run without `--env
  default:
    # Everything from the values.yaml is available via `{{ .Values.KEY }}`.
    # Suppose `{"foo": {"bar": 1}}` contained in the values.yaml below,
    # `{{ .Values.foo.bar }}` is evaluated to `1`.
    values:
    - environments/default/values.yaml
    # Each entry in values can be either a file path or inline values.
    # The below is an example of inline values, which is merged to the `.Values`
    - myChartVer: 1.0.0-dev
  # Any environment other than `default` is used only when `helmfile` is run with `--envi
  # That is, the "production" env below is used when and only when it is run like `helmfi
  production:
    values:
    - environments/production/values.yaml
    - myChartVer: 1.0.0
    # disable vault release processing
    - vault:
        enabled: false
    ## `secrets.yaml` is decrypted by `helm-secrets` and available via `{{ .Environment.V
    secrets:
    - environments/production/secrets.yaml
    # Instructs helmfile to fail when unable to find a environment values file listed und
    #
    # Possible values are  "Error", "Warn", "Info", "Debug". The default is "Error".
    #
    # Use "Warn", "Info", or "Debug" if you want helmfile to not fail when a values file
    # a message about the missing file at the log-level.
    missingFileHandler: Error
    # kubeContext to use for this environment
    kubeContext: kube-context

#
# Advanced Configuration: Layering
#
# Helmfile merges all the "base" state files and this state file before processing.
#
# Assuming this state file is named `helmfile.yaml`, all the files are merged in the orde
#   environments.yaml <- defaults.yaml <- templates.yaml <- helmfile.yaml
bases:
- environments.yaml
- defaults.yaml
- templates.yaml

#
# Advanced Configuration: API Capabilities
#
# 'helmfile template' renders releases locally without querying an actual cluster,
# and in this case `.Capabilities.APIVersions` cannot be populated.
# When a chart queries for a specific CRD or the Kubernetes version, this can lead to une
#
# Note that `Capabilities.KubeVersion` is deprecated in Helm 3 and `helm template` won't
# All you can do is fix your chart to respect `.Capabilities.APIVersions` instead, rather
# how to set `Capabilities.KubeVersion` in Helmfile.
#
# Configure a fixed list of API versions to pass to 'helm template' via the --api-version
apiVersions:
- example/v1

# DEPRECATED: This is available only on Helm 2, which has been EOL since 2020
# Configure a Kubernetes version to  pass to 'helm template' via the --kube-version flag:
# See https://github.com/roboll/helmfile/pull/2002 for more information.
kubeVersion: v1.21
```

Templating

Helmfile uses Go templates for templating your helmfile.yaml. While go ships several built-in functions, we have added all of the functions in the Sprig library.

We also added the following functions:

- `requiredEnv`
- `exec`
- `envExec`
- `readFile`
- `readDir`
- `readDirEntries`
- `toYaml`
- `fromYaml`
- `setValueAtPath`
- `get` (Sprig's original `get` is available as `sprigGet`)
- `tpl`
- `required`
- `fetchSecretValue`
- `expandSecretRefs`

More details on each function can be found at "Template Functions" page in our documentation.

Using environment variables

Environment variables can be used in most places for templating the helmfile. Currently this is supported for `name`, `namespace`, `value` (in set), `values` and `url` (in repositories).

Examples:

```
repositories:
- name: your-private-git-repo-hosted-charts
  url: https://{{ requiredEnv "GITHUB_TOKEN"}}@raw.githubusercontent.com/kmzfs/helm-repo-
```

```
releases:
  - name: {{ requiredEnv "NAME" }}-vault
    namespace: {{ requiredEnv "NAME" }}
    chart: roboll/vault-secret-manager
    values:
      - db:
          username: {{ requiredEnv "DB_USERNAME" }}
          password: {{ requiredEnv "DB_PASSWORD" }}
    set:
      - name: proxy.domain
        value: {{ requiredEnv "PLATFORM_ID" }}.my-domain.com
      - name: proxy.scheme
        value: {{ env "SCHEME" | default "https" }}
```

Note

If you wish to treat your enviroment variables as strings always, even if they are boolean or numeric values you can use `{{ env "ENV_NAME" | quote }}` or `"{{ env "ENV_NAME" }}"`. These approaches also work with `requiredEnv`.

```
Declaratively deploy your Kubernetes manifests, Kustomize configs, and Charts as Helm rel

Usage:
  helmfile [command]

Available Commands:
  apply        Apply all resources from state file only when there are changes
  build        Build all resources from state file
  cache        Cache management
  charts       DEPRECATED: sync releases from state file (helm upgrade --install)
  completion   Generate the autocompletion script for the specified shell
  delete       DEPRECATED: delete releases from state file (helm delete)
  deps         Update charts based on their requirements
  destroy      Destroys and then purges releases
  diff         Diff releases defined in state file
  fetch        Fetch charts from state file
  help         Help about any command
  init         Initialize the helmfile, includes version checking and installation of hel
  lint         Lint charts from state file (helm lint)
  list         List releases defined in state file
  repos        Repos releases defined in state file
  status       Retrieve status of releases in state file
  sync         Sync releases defined in state file
  template     Template releases defined in state file
  test         Test charts from state file (helm test)
  version      Print the CLI version
  write-values Write values files for releases. Similar to `helmfile template`, write val

Flags:
      --allow-no-matching-release       Do not exit with an error code if the provided se
  -c, --chart string                    Set chart. Uses the chart set in release by defau
      --color                           Output with color
      --debug                           Enable verbose output for Helm and set log-level
      --enable-live-output              Show live output from the Helm binary Stdout/Stde
                                        It only applies for the Helm CLI commands, Stdout
  -e, --environment string              specify the environment name. defaults to "defaul
  -f, --file helmfile.yaml              load config from file or directory. defaults to h
  -b, --helm-binary string              Path to the helm binary (default "helm")
  -h, --help                            help for helmfile
  -i, --interactive                     Request confirmation before attempting to modify
      --kube-context string             Set kubectl context. Uses current context by defa
      --log-level string                Set log level, default info (default "info")
  -n, --namespace string                Set namespace. Uses the namespace set in the cont
      --no-color                        Output without color
  -q, --quiet                           Silence output. Equivalent to log-level warn
  -l, --selector stringArray            Only run using the releases that match labels. La
                                        A release must match all labels in a group in ord
                                        "--selector tier=frontend,tier!=proxy --selector
                                        The name of a release can be used as a label: "--
      --state-values-file stringArray   specify state values in a YAML file
      --state-values-set stringArray    set state values on the command line (can specify
  -v, --version                         version for helmfile

Use "helmfile [command] --help" for more information about a command.
```

init

The `helmfile init` sub-command checks the dependencies required for helmfile operation, such as `helm`, `helm diff plugin`, `helm secrets plugin`, `helm helm-git plugin`, `helm s3 plugin`. When it does not exist or the version is too low, it can be installed automatically.

The `helmfile cache` sub-command is designed for cache management. Go-getter-backed remote file system are cached by `helmfile`. There is no TTL implemented, if you need to update the cached files or directories, you need to clean individually or run a full cleanup with `helmfile cache cleanup`

sync

The `helmfile sync` sub-command sync your cluster state as described in your `helmfile`. The default helmfile is `helmfile.yaml`, but any YAML file can be passed by specifying a `--file path/to/your/yaml/file` flag.

Under the covers, Helmfile executes `helm upgrade --install` for each `release` declared in the manifest, by optionally decrypting secrets to be consumed as helm chart values. It also updates specified chart repositories and updates the dependencies of any referenced local charts.

For Helm 2.9+ you can use a username and password to authenticate to a remote repository.

deps

The `helmfile deps` sub-command locks your helmfile state and local charts dependencies.

It basically runs `helm dependency update` on your helmfile state file and all the referenced local charts, so that you get a "lock" file per each helmfile state or local chart.

All the other `helmfile` sub-commands like `sync` use chart versions recorded in the lock files, so that e.g. untested chart versions won't suddenly get deployed to the production environment.

For example, the lock file for a helmfile state file named `helmfile.1.yaml` will be `helmfile.1.lock`. The lock file for a local chart would be `requirements.lock`, which is the same as `helm`.

The lock file can be changed using `lockFilePath` in helm state, which makes it possible to for example have a different lock file per environment via templating.

It is recommended to version-control all the lock files, so that they can be used in the production deployment pipeline for extra reproducibility.

To bring in chart updates systematically, it would also be a good idea to run `helmfile` regularly, test it, and then update the lock files in the version-control system.

diff

The `helmfile diff` sub-command executes the helm-diff plugin across all of the charts/releases defined in the manifest.

To supply the diff functionality Helmfile needs the helm-diff plugin v2.9.0+1 or greater installed. For Helm 2.3+
you should be able to simply execute `helm plugin install https://github.com/databus23/helm-diff`. For more details please look at their documentation.

apply

The `helmfile apply` sub-command begins by executing `diff`. If `diff` finds that there is any changes, `sync` is executed. Adding `--interactive` instructs Helmfile to request your confirmation before `sync`.

An expected use-case of `apply` is to schedule it to run periodically, so that you can auto-fix skews between the desired and the current state of your apps running on Kubernetes clusters.

destroy

The `helmfile destroy` sub-command uninstalls and purges all the releases defined in the manifests.

`helmfile --interactive destroy` instructs Helmfile to request your confirmation before actually deleting releases.

`destroy` basically runs `helm uninstall --purge` on all the targeted releases. If you don't want purging, use `helmfile delete` instead.

delete (DEPRECATED)

The `helmfile delete` sub-command deletes all the releases defined in the manifests.

`helmfile --interactive delete` instructs Helmfile to request your confirmation before actually deleting releases.

Note that `delete` doesn't purge releases. So `helmfile delete && helmfile sync` results in sync failed due to that releases names are not deleted but preserved for future refere[illegible] If you really want to remove releases for reuse, add `--purge` flag to run it like `helmfile [illegible] --purge`.

secrets

The `secrets` parameter in a `helmfile.yaml` causes the helm-secrets plugin to be executed to decrypt the file.

To supply the secret functionality Helmfile needs the `helm secrets` plugin installed. For Helm 2.3+
you should be able to simply execute `helm plugin install https://github.com/jkroepke/helm-secrets`.

test

The `helmfile test` sub-command runs a `helm test` against specified releases in the manifest, default to all

Use `--cleanup` to delete pods upon completion.

lint

The `helmfile lint` sub-command runs a `helm lint` across all of the charts/releases defined in the manifest. Non local charts will be fetched into a temporary folder which will be deleted once the task is completed.

fetch

The `helmfile fetch` sub-command downloads or copies local charts to a local directory for debug purpose. The local directory
must be specified with `--output-dir`.

list

The `helmfile list` sub-command lists releases defined in the manifest. Optional `--output` flag accepts `json` to output releases in JSON format.

If `--skip-charts` flag is not set, list would prepare all releases, by fetching charts and templating them.

version

The `helmfile version` sub-command prints the version of Helmfile.Optional `-o` flag accepts `json` `yaml` `short` to output version in JSON, YAML or short format.

default it will check for the latest version of Helmfile and print a tip if the current version is not the latest. To disable this behavior, set environment variable `HELMFILE_UPGRADE_NOTICE_DISABLED` to any non-empty value.

Paths Overview

Using manifest files in conjunction with command line argument can be a bit confusing.

A few rules to clear up this ambiguity:

- Absolute paths are always resolved as absolute paths
- Relative paths referenced *in* the Helmfile manifest itself are relative to that manifest
- Relative paths referenced on the command line are relative to the current working directory the user is in
- Relative paths referenced from within the helmfile loaded from the standard input using `helmfile -f -` are relative to the current working directory

For additional context, take a look at paths examples.

Labels Overview

A selector can be used to only target a subset of releases when running Helmfile. This is useful for large helmfiles with releases that are logically grouped together.

Labels are simple key value pairs that are an optional field of the release spec. When selecting by label, the search can be inverted. `tier!=backend` would match all releases that do NOT have the `tier: backend` label. `tier=fronted` would only match releases with the `tier: frontend` label.

Multiple labels can be specified using `,` as a separator. A release must match all selectors in order to be selected for the final helm command.

The `selector` parameter can be specified multiple times. Each parameter is resolved independently so a release that matches any parameter will be used.

`--selector tier=frontend --selector tier=backend` will select all the charts.

In addition to user supplied labels, the name, the namespace, and the chart are available to be used as selectors. The chart will just be the chart name excluding the repository (Example `stable/filebeat` would be selected using `--selector chart=filebeat`).

`commonLabels` can be used when you want to apply the same label to all releases and use templating based on that.
For instance, you install a number of charts on every customer but need to provide different values file per customer.

templates/common.yaml:

```
templates:
  nginx: &nginx
    name: nginx
    chart: stable/nginx-ingress
    values:
    - ../values/common/{{ .Release.Name }}.yaml
    - ../values/{{ .Release.Labels.customer }}/{{ .Release.Name }}.yaml

  cert-manager: &cert-manager
    name: cert-manager
    chart: jetstack/cert-manager
    values:
    - ../values/common/{{ .Release.Name }}.yaml
    - ../values/{{ .Release.Labels.customer }}/{{ .Release.Name }}.yaml
```

helmfile.yaml:

```
{{ readFile "templates/common.yaml" }}

commonLabels:
  customer: company

releases:
- <<: *nginx
- <<: *cert-manager
```

Templates

You can use go's text/template expressions in `helmfile.yaml` and `values.yaml.gotmpl` (templated helm values files). `values.yaml` references will be used verbatim. In other words:

- for value files ending with `.gotmpl`, template expressions will be rendered
- for plain value files (ending in `.yaml`), content will be used as-is

In addition to built-in ones, the following custom template functions are available:

- `readFile` reads the specified local file and generate a golang string
- `readDir` reads the files within provided directory path. (folders are excluded)
- `readDirEntries` Returns a list of https://pkg.go.dev/os#DirEntry within provided directory path
- `fromYaml` reads a golang string and generates a map
- `setValueAtPath PATH NEW_VALUE` traverses a golang map, replaces the value at the PATH with NEW_VALUE
- `toYaml` marshals a map into a string

- `get` returns the value of the specified key if present in the `.Values` object, otherwise will return the default value defined in the function

Values Files Templates

You can reference a template of values file in your `helmfile.yaml` like below:

```
releases:
- name: myapp
  chart: mychart
  values:
  - values.yaml.gotmpl
```

Every values file whose file extension is `.gotmpl` is considered as a template file.

Suppose `values.yaml.gotmpl` was something like:

```
{{ readFile "values.yaml" | fromYaml | setValueAtPath "foo.bar" "FOO_BAR" | toYaml }}
```

And `values.yaml` was:

```
foo:
  bar: ""
```

The resulting, temporary values.yaml that is generated from `values.yaml.gotmpl` would become:

```
foo:
  # Notice `setValueAtPath "foo.bar" "FOO_BAR"` in the template above
  bar: FOO_BAR
```

Refactoring `helmfile.yaml` with values files templates

One of expected use-cases of values files templates is to keep `helmfile.yaml` small and concise.

See the example `helmfile.yaml` below:

```
releases:
  - name: {{ requiredEnv "NAME" }}-vault
    namespace: {{ requiredEnv "NAME" }}
    chart: roboll/vault-secret-manager
    values:
      - db:
          username: {{ requiredEnv "DB_USERNAME" }}
          password: {{ requiredEnv "DB_PASSWORD" }}
    set:
      - name: proxy.domain
        value: {{ requiredEnv "PLATFORM_ID" }}.my-domain.com
      - name: proxy.scheme
        value: {{ env "SCHEME" | default "https" }}
```

The `values` and `set` sections of the config file can be separated out into a template:

`helmfile.yaml`:

```
releases:
  - name: {{ requiredEnv "NAME" }}-vault
    namespace: {{ requiredEnv "NAME" }}
    chart: roboll/vault-secret-manager
    values:
    - values.yaml.gotmpl
```

`values.yaml.gotmpl`:

```
db:
  username: {{ requiredEnv "DB_USERNAME" }}
  password: {{ requiredEnv "DB_PASSWORD" }}
proxy:
  domain: {{ requiredEnv "PLATFORM_ID" }}.my-domain.com
  scheme: {{ env "SCHEME" | default "https" }}
```

Environment

When you want to customize the contents of `helmfile.yaml` or `values.yaml` files per environment, use this feature.

You can define as many environments as you want under `environments` in `helmfile.yaml`.

The environment name defaults to `default`, that is, `helmfile sync` implies the `default` environment.
The selected environment name can be referenced from `helmfile.yaml` and `values.yaml.gotmpl` by `{{ .Environment.Name }}`.

If you want to specify a non-default environment, provide a `--environment NAME` flag to `helmfile` like `helmfile --environment production sync`.

The below example shows how to define a production-only release:

```
environments:
  default:
  production:

---

releases:
- name: newrelic-agent
  installed: {{ eq .Environment.Name "production" | toYaml }}
  # snip
- name: myapp
  # snip
```

Environment Values

Environment Values allows you to inject a set of values specific to the selected environment, into values.yaml templates.
Use it to inject common values from the environment to multiple values files, to make your configuration DRY.

Suppose you have three files `helmfile.yaml`, `production.yaml` and `values.yaml.gotmpl`:

`helmfile.yaml`

```
environments:
  production:
    values:
    - production.yaml

---

releases:
- name: myapp
  values:
  - values.yaml.gotmpl
```

`production.yaml`

```
domain: prod.example.com
releaseName: prod
```

`values.yaml.gotmpl`

```
domain: {{ .Values | get "domain" "dev.example.com" }}
```

`helmfile sync` installs `myapp` with the value `domain=dev.example.com`, whereas `helmfile --environment production sync` installs the app with the value `domain=prod.example.com`.

For even more flexibility, you can now use values declared in the `environments:` sect v: latest other parts of your helmfiles:

consider:

`default.yaml`

```
domain: dev.example.com
releaseName: dev
```

```
environments:
  default:
    values:
    - default.yaml
  production:
    values:
    - production.yaml #  bare .yaml file, content will be used verbatim
    - other.yaml.gotmpl  #  template directives with potential side-effects like `exec` a

---

releases:
- name: myapp-{{ .Values.releaseName }} # release name will be one of `dev` or `prod` dep
  values:
  - values.yaml.gotmpl
- name: production-specific-release
  # this release would be installed only if selected environment is `production`
  installed: {{ eq .Values.releaseName "prod" | toYaml }}
  ...
```

Note on Environment.Values vs Values

The `{{ .Values.foo }}` syntax is the recommended way of using environment values.

Prior to this pull request, environment values were made available through the `{{ .Environment.Values.foo }}` syntax.
This is still working but is **deprecated** and the new `{{ .Values.foo }}` syntax should be used instead.

You can read more infos about the feature proposal here.

Loading remote Environment values files

Since Helmfile v0.118.8, you can use `go-getter`-style URLs to refer to remote values files:

```
environments:
  cluster-azure-us-west:
    values:
      - git::https://git.company.org/helmfiles/global/azure.yaml?ref=master
      - git::https://git.company.org/helmfiles/global/us-west.yaml?ref=master
      - git::https://gitlab.com/org/repository-name.git@/config/config.test.yaml?ref=main
  cluster-gcp-europe-west:
    values:
      - git::https://git.company.org/helmfiles/global/gcp.yaml?ref=master
      - git::https://git.company.org/helmfiles/global/europe-west.yaml?ref=master
      - git::https://ci:{{ env "CI_JOB_TOKEN" }}@gitlab.com/org/repository-name.git@/conf
  staging:
     values:
      - git::https://{{ env "GITHUB_PAT" }}@github.com/[$GITHUB_ORGorGITHUB_USER]/reposit
      - http://$HOSTNAME/artifactory/example-repo-local/test.tgz@values.yaml #Artifactory
---

releases:
  - ...
```

For more information about the supported protocols see: go-getter Protocol-Specific Options.

This is particularly useful when you co-locate helmfiles within your project repo but want to reuse the definitions in a global repo.

Environment Secrets

Environment Secrets *(not to be confused with Kubernetes Secrets)* are encrypted versions of `Environment Values`.
You can list any number of `secrets.yaml` files created using `helm secrets` or `sops`, so that Helmfile could automatically decrypt and merge the secrets into the environment values.

First you must have the helm-secrets plugin installed along with a `.sops.yaml` file to configure the method of encryption (this can be in the same directory as your helmfile or
in the subdirectory containing your secrets files).

Then suppose you have a secret `foo.bar` defined in `environments/production/secrets.yaml`:

```
foo.bar: "mysupersecretstring"
```

You can then encrypt it with `helm secrets enc environments/production/secrets.yaml`

Then reference that encrypted file in `helmfile.yaml`:

```
environments:
  production:
    secrets:
    - environments/production/secrets.yaml

---

releases:
- name: myapp
  chart: mychart
  values:
  - values.yaml.gotmpl
```

Then the environment secret `foo.bar` can be referenced by the below template expression in your `values.yaml.gotmpl`:

```
{{ .Values.foo.bar }}
```

Loading remote Environment secrets files

Since Helmfile v0.149.0, you can use `go-getter`-style URLs to refer to remote secrets files, the same way as in values files:

```
environments:
  staging:
    secrets:
      - git::https://{{ env "GITHUB_PAT" }}@github.com/org/repo.git@/environments/staging
      - http://$HOSTNAME/artifactory/example-repo-local/test.tgz@environments/staging.sec
  production:
    secrets:
      - git::https://{{ env "GITHUB_PAT" }}@github.com/org/repo.git@/environments/product
      - http://$HOSTNAME/artifactory/example-repo-local/test.tgz@environments/production.
```

DAG-aware installation/deletion ordering with `needs`

`needs` controls the order of the installation/deletion of the release:

```
releases:
- name: somerelease
  needs:
  - [[KUBECONTEXT/]NAMESPACE/]anotherelease
```

Be aware that you have to specify the kubecontext and namespace name if you configured one for the release(s).

All the releases listed under `needs` are installed before(or deleted after) the release itself.

For the following example, `helmfile [sync|apply]` installs releases in this order:

1. logging
2. servicemesh

3. myapp1 and myapp2

```
  - name: myapp1
    chart: charts/myapp
    needs:
    - servicemesh
    - logging
  - name: myapp2
    chart: charts/myapp
    needs:
    - servicemesh
    - logging
  - name: servicemesh
    chart: charts/istio
    needs:
    - logging
  - name: logging
    chart: charts/fluentd
```

Note that all the releases in a same group is installed concurrently. That is, myapp1 and myapp2 are installed concurrently.

On `helmfile [delete|destroy]`, deletions happen in the reverse order.

That is, `myapp1` and `myapp2` are deleted first, then `servicemesh`, and finally `logging`.

Selectors and `needs`

When using selectors/labels, `needs` are ignored by default. This behaviour can be overruled with a few parameters:
Parameter	default	Description
—	—	—
`--skip-needs`	`true`	`needs` are ignored (default behavior).
`--include-needs`	`false`	The direct `needs` of the selected release(s) will be included.
`--include-transitive-needs`	`false`	The direct and transitive `needs` of the selected release(s) will be included.
Let's look at an example to illustrate how the different parameters work:

```
releases:
- name: serviceA
  chart: my/chart
  needs:
  - serviceB
- name: serviceB
  chart: your/chart
  needs:
  - serviceC
- name: serviceC
  chart: her/chart
- name: serviceD
  chart: his/chart
```

Command	Included Releases Order	Explanation
`helmfile -l name=serviceA sync`	- `serviceA`	By default no needs are included.
`helmfile -l name=serviceA sync --include-needs`	- `serviceB` - `serviceA`	`serviceB` is now part of the release as it is a direct need of `se`
`helmfile -l name=serviceA sync --include-transitive-needs`	- `serviceC` - `serviceB` - `serviceA`	`serviceC` is now also part of the release as it is a direct need c and therefore a transitive need of `serviceA`.

Note that `--include-transitive-needs` will override any potential exclusions done by selectors or conditions. So even if you explicitly exclude a release via a selector it will still be part of the deployment in case it is a direct or transitive need of any of the specified releases.

Separating helmfile.yaml into multiple independent files

Once your `helmfile.yaml` got to contain too many releases,
split it into multiple yaml files.

Recommended granularity of helmfile.yaml files is “per microservice” or “per team”.
And there are two ways to organize your files.

- Single directory
- Glob patterns

Single directory

`helmfile -f path/to/directory` loads and runs all the yaml files under the specified directory, each file as an independent helmfile.yaml.
The default helmfile directory is `helmfile.d`, that is,
in case helmfile is unable to locate `helmfile.yaml`, it tries to locate `helmfile.d/*.yaml`.

All the yaml files under the specified directory are processed in the alphabetical order. For example, you can use a `<two digit number>-<microservice>.yaml` naming convention to control the sync order.

- `helmfile.d`/

- `00-database.yaml`
- `00-backend.yaml`
- `01-frontend.yaml`

Glob patterns

In case you want more control over how multiple `helmfile.yaml` files are organized, use `helmfiles:` configuration key in the `helmfile.yaml`:

Suppose you have multiple microservices organized in a Git repository that looks like:

- `myteam/` (sometimes it is equivalent to a k8s ns, that is `kube-system` for `clusterops` team)
- `apps/`
  - `filebeat/`
  - `helmfile.yaml` (no `charts/` exists, because it depends on the stable/filebeat chart hosted on the official helm charts repository)
  - `README.md` (each app managed by my team has a dedicated README maintained by the owners of the app)
  - `metricbeat/`
  - `helmfile.yaml`
  - `README.md`
  - `elastalert-operator/`
  - `helmfile.yaml`
  - `README.md`
  - `charts/`
    - `elastalert-operator/`
    - `<the content of the local helm chart>`

The benefits of this structure is that you can run `git diff` to locate in which directory=microservice a git commit has changes.
It allows your CI system to run a workflow for the changed microservice only.

A downside of this is that you don't have an obvious way to sync all microservices at once. That is, you have to run:

```
for d in apps/*; do helmfile -f $d diff; if [ $? -eq 2 ]; then helmfile -f $d sync; fi, u
```

At this point, you'll start writing a `Makefile` under `myteam/` so that `make sync-all` w

the job.

It does work, but you can rely on the Helmfile feature instead.

Put `myteam/helmfile.yaml` that looks like:

```
helmfiles:
- apps/*/helmfile.yaml
```

So that you can get rid of the `Makefile` and the bash snippet.
Just run `helmfile sync` inside `myteam/`, and you are done.

All the files are sorted alphabetically per group = array item inside `helmfiles:`, so that you have granular control over ordering, too.

selectors

When composing helmfiles you can use selectors from the command line as well as explicit selectors inside the parent helmfile to filter the releases to be used.

```
helmfiles:
- apps/*/helmfile.yaml
- path: apps/a-helmfile.yaml
  selectors:          # list of selectors
  - name=prometheus
  - tier=frontend
- path: apps/b-helmfile.yaml # no selector, so all releases are used
selectors: []
- path: apps/c-helmfile.yaml # parent selector to be used or cli selector for the initial
  selectorsInherited: true
```

- When a selector is specified, only this selector applies and the parents or CLI selectors are ignored.
- When not selector is specified there are 2 modes for the selector inheritance because we would like to change the current inheritance behavior (see issue #344).
- Legacy mode, sub-helmfiles without selectors inherit selectors from their parent helmfile. The initial helmfiles inherit from the command line selectors.
- explicit mode, sub-helmfile without selectors do not inherit from their parent or the CLI selector. If you want them to inherit from their parent selector then use `selectorsInherited: true`. To enable this explicit mode you need to set the following environment variable `HELMFILE_EXPERIMENTAL=explicit-selector-inheritance` (see experimental).
- Using `selector: []` will select all releases regardless of the parent selector or cli for the initial helmfile
- using `selectorsInherited: true` make the sub-helmfile selects releases with the parent selector or the cli for the initial helmfile. You cannot specify an explicit selector while using `selectorsInherited: true`

Importing values from any source

The `exec` template function that is available in `values.yaml.gotmpl` is useful for importing values from any source
that is accessible by running a command:

A usual usage of `exec` would look like this:

```
mysetting: |
{{ exec "./mycmd" (list "arg1" "arg2" "--flag1") | indent 2 }}
```

Or even with a pipeline:

```
mysetting: |
{{ yourinput | exec "./mycmd-consume-stdin" (list "arg1" "arg2") | indent 2 }}
```

The possibility is endless. Try importing values from your golang app, bash script, jsonnet, or anything!

Then `envExec` same as `exec`, but it can receive a dict as the envs.

A usual usage of `envExec` would look like this:

```
mysetting: |
{{ envExec (dict "envkey" "envValue") "./mycmd" (list "arg1" "arg2" "--flag1") | indent 2
```

Hooks

A Helmfile hook is a per-release extension point that is composed of:

- `events`
- `command`
- `args`
- `showlogs`

Helmfile triggers various `events` while it is running.
Once `events` are triggered, associated `hooks` are executed, by running the `command` with `args`. The standard output of the `command` will be displayed if `showlogs` is set and it's value is `true`.

Currently supported `events` are:

- `prepare`

- `preapply`
- `presync`
- `preuninstall`
- `postuninstall`
- `postsync`
- `cleanup`

Hooks associated to `prepare` events are triggered after each release in your helmfile is loaded from YAML, before execution.
`prepare` hooks are triggered on the release as long as it is not excluded by the helmfile selector(e.g. `helmfile -l key=value`).

Hooks associated to `presync` events are triggered before each release is synced (installed or upgraded) on the cluster.
This is the ideal event to execute any commands that may mutate the cluster state as it will not be run for read-only operations like `lint`, `diff` or `template`.

`preapply` hooks are triggered before a release is uninstalled, installed, or upgraded as part of `helmfile apply`.
This is the ideal event to hook into when you are going to use `helmfile apply` for every kind of change and you want the hook to be triggered regardless of whether the releases have changed or not. Be sure to make each `preapply` hook command idempotent. Otherwise, rerunning helmfile-apply on a transient failure may end up either breaking your cluster, or the hook that runs for the second time will never succeed.

`preuninstall` hooks are triggered immediately before a release is uninstalled as part of `helmfile apply`, `helmfile sync`, `helmfile delete`, and `helmfile destroy`.

`postuninstall` hooks are triggered immediately after successful uninstall of a release while running `helmfile apply`, `helmfile sync`, `helmfile delete`, `helmfile destroy`.

`postsync` hooks are triggered after each release is synced (installed or upgraded) on the cluster, regardless if the sync was successful or not.
This is the ideal place to execute any commands that may mutate the cluster state as it will not be run for read-only operations like `lint`, `diff` or `template`.

`cleanup` hooks are triggered after each release is processed.
This is the counterpart to `prepare`, as any release on which `prepare` has been triggered gets `cleanup` triggered as well.

The following is an example hook that just prints the contextual information provide
hook:

```
releases:
- name: myapp
  chart: mychart
  # *snip*
  hooks:
  - events: ["prepare", "cleanup"]
    showlogs: true
    command: "echo"
    args: ["{{`{{.Environment.Name}}`}}", "{{`{{.Release.Name}}`}}", "{{`{{.HelmfileComma
"]
```

Let's say you ran `helmfile --environment prod sync`, the above hook results in executing:

```
echo {{Environment.Name}} {{.Release.Name}} {{.HelmfileCommand}}
```

Whereas the template expressions are executed thus the command becomes:

```
echo prod myapp sync
```

Now, replace `echo` with any command you like, and rewrite `args` that actually conforms to the command, so that you can integrate any command that does:

- templating
- linting
- testing

Hooks expose additional template expressions:

`.Event.Name` is the name of the hook event.

`.Event.Error` is the error generated by a failed release, exposed for `postsync` hooks only when a release fails, otherwise its value is `nil`.

You can use the hooks event expressions to send notifications to platforms such as `Slack`, `MS Teams`, etc.

The following example passes arguments to a script which sends a notification:

```
releases:
- name: myapp
  chart: mychart
  # *snip*
  hooks:
  - events:
    - presync
    - postsync
    showlogs: true
    command: notify.sh
    args:
    - --event
    - '{{`{{ .Event.Name }}`}}'
    - --status
    - '{{`{{ if .Event.Error }}failure{{ else }}success{{ end }}`}}'
    - --environment
    - '{{`{{ .Environment.Name }}`}}'
    - --namespace
    - '{{`{{ .Release.Namespace }}`}}'
    - --release
    - '{{`{{ .Release.Name }}`}}'
```

For templating, imagine that you created a hook that generates a helm chart on-the-fly by running an external tool like ksonnet, kustomize, or your own template engine.
It will allow you to write your helm releases with any language you like, while still leveraging goodies provided by helm.

Global Hooks

In contrast to the per release hooks mentioned above these are run only once at the very beginning and end of the execution of a helmfile command and only the `prepare` and `cleanup` hooks are available respectively.

They use the same syntax as per release hooks, but at the top level of your helmfile:

```
hooks:
- events: ["prepare", "cleanup"]
  showlogs: true
  command: "echo"
  args: ["{{`{{.Environment.Name}}`}}", "{{`{{.HelmfileCommand}}`}}\
"]
```

Helmfile + Kustomize

Do you prefer `kustomize` to write and organize your Kubernetes apps, but still want to leverage helm's useful features
like rollback, history, and so on? This section is for you!

The combination of `hooks` and helmify-kustomize
enables you to integrate kustomize into Helmfile.

That is, you can use `kustomize` to build a local helm chart from a kustomize overlay. v: latest

Let's assume you have a kustomize project named `foo-kustomize` like this:

```
foo-kustomize/
├── base
│   ├── configMap.yaml
│   ├── deployment.yaml
│   ├── kustomization.yaml
│   └── service.yaml
└── overlays
    ├── default
    │   ├── kustomization.yaml
    │   └── map.yaml
    ├── production
    │   ├── deployment.yaml
    │   └── kustomization.yaml
    └── staging
        ├── kustomization.yaml
        └── map.yaml

5 directories, 10 files
```

Write `helmfile.yaml`:

```
- name: kustomize
  chart: ./foo
  hooks:
  - events: ["prepare", "cleanup"]
    command: "../helmify"
    args: ["{{`{{if eq .Event.Name \"prepare\"}}build{{else}}clean{{end}}`}}", "{{`{{.Rel
art}}`}}", "{{`{{.Environment.Name}}`}}"]
```

Run `helmfile --environment staging sync` and see it results in helmfile running `kustomize build foo-kustomize/overlays/staging > foo/templates/all.yaml`.

Voilà! You can mix helm releases that are backed by remote charts, local charts, and even kustomize overlays.

Guides

Use the Helmfile Best Practices Guide to write advanced helmfiles that feature:

- Default values
- Layering

We also have dedicated documentation on the following topics which might interest you:

- Shared Configurations Across Teams

Or join our friendly slack community in the `#helmfile` channel to ask questions and get help. Check out our slack archive for good examples of how others are using it.

Using .env files

Helmfile itself doesn't have an ability to load .env files. But you can write some bash script to achieve the goal:

```
set -a; . .env; set +a; helmfile sync
```

Please see #203 for more context.

Running Helmfile interactively

`helmfile --interactive [apply|destroy|delete|sync]` requests confirmation from you before actually modifying your cluster.

Use it when you're running `helmfile` manually on your local machine or a kind of secure administrative hosts.

For your local use-case, aliasing it like `alias hi='helmfile --interactive'` would be convenient.

Running Helmfile without an Internet connection

Once you download all required charts into your machine, you can run `helmfile charts` to deploy your apps.
It basically run only `helm upgrade --install` with your already-downloaded charts, hence no Internet connection is required.
See #155 for more information on this topic.

Experimental Features

Some experimental features may be available for testing in perspective of being (or not) included in a future release.
Those features are set using the environment variable `HELMFILE_EXPERIMENTAL`. Here is the current experimental feature :

- `explicit-selector-inheritance` : remove today implicit cli selectors inheritance for composed helmfiles, see composition selector

If you want to enable all experimental features set the env var to `HELMFILE_EXPERIMENTAL=true`

`bash` and `zsh` completion

helmfile completion –help

Examples

For more examples, see the examples/README.md or the `helmfile` distribution by Cloud Posse.

Integrations

- renovate automates chart version updates. See this PR for more information.
- For updating container image tags and git tags embedded within helmfile.yaml and values, you can use renovate's regexManager. Please see this comment in the renovate repository for more information.
- ArgoCD Integration
- Azure ACR Integration

ArgoCD Integration

Use ArgoCD with `helmfile template` for GitOps.

ArgoCD has support for kustomize/manifests/helm chart by itself. Why bother with Helmfile?

The reasons may vary:

1. You do want to manage applications with ArgoCD, while letting Helmfile manage infrastructure-related components like Calico/Cilium/WeaveNet, Linkerd/Istio, and ArgoCD itself.

- This way, any application deployed by ArgoCD has access to all the infrastructure.
- Of course, you can use ArgoCD's Sync Waves and Phases for ordering the infrastructure and application installations. But it may be difficult to separate the concern between the infrastructure and apps and annotate K8s resources consistently when you have different teams for managing infra and apps.

2. You want to review the exact K8s manifests being applied on pull-request time, before ArgoCD syncs.

- This is often better than using a kind of `HelmRelease` custom resources that obfuscates exactly what manifests are being applied, which makes reviewing harder.

3. Use Helmfile as the single-pane of glass for all the K8s resources deployed to your cluster(s).

- Helmfile can reduce repetition in K8s manifests across ArgoCD application

For 1, you run `helmfile apply` on CI to deploy ArgoCD and the infrastructure components.

> helmfile config for this phase often reside within the same directory as your Terraform project. So connecting the two with terraform-provider-helmfile may be helpful

For 2, another app-centric CI or bot should render/commit manifests by running:

```
helmfile template --output-dir-template $(pwd)/gitops//{{.Release.Name}}
cd gitops
git add .
git commit -m 'some message'
git push origin $BRANCH
```

> Note that `$(pwd)` is necessary when `helmfile.yaml` has one or more sub-helmfiles in nested directories,
> because setting a relative file path in `--output-dir` or `--output-dir-template` results in each sub-helmfile render
> to the directory relative to the specified path.

so that they can be deployed by Argo CD as usual.

The CI or bot can optionally submit a PR to be review by human, running:

```
hub pull-request -b main -l gitops -m 'some description'
```

Recommendations:

- Do create ArgoCD `Application` custom resource per Helm/Helmfile release, each point to respective sub-directory generated by `helmfile template --output-dir-template`
- If you don't directly push it to the main Git branch and instead go through a pull-request, do lint rendered manifests on your CI, so that you can catch easy mistakes earlier/before ArgoCD finally deploys it
- See this ArgoCD issue for why you may want this, and see this helmfile issue for how `--output-dir-template` works.

Azure ACR Integration

Azure offers helm repository support for Azure Container Registry as a preview featu

To use this you must first `az login` and then `az acr helm repo add -n <MyRegistry>`. This will extract a token for the given ACR and configure `helm` to use it, e.g. `helm repo update` should work straight away.

To use `helmfile` with ACR, on the other hand, you must either include a username/password in the repository definition for the ACR in your `helmfile.yaml` or use the `--skip-deps` switch, e.g. `helmfile template --skip-deps`.

An ACR repository definition in `helmfile.yaml` looks like this:

```
repositories:
  - name: <MyRegistry>
    url: https://<MyRegistry>.azurecr.io/helm/v1/repo
```

OCI Registries

In order to use OCI chart registries firstly they must be marked in the repository list as OCI enabled, e.g.

```
repositories:
  - name: myOCIRegistry
    url: myregistry.azurecr.io
    oci: true
```

It is important not to include a scheme for the URL as helm requires that these are not present for OCI registries

Secondly the credentials for the OCI registry can either be specified within `helmfile.yaml` similar to

```
repositories:
  - name: myOCIRegistry
    url: myregistry.azurecr.io
    oci: true
    username: spongebob
    password: squarepants
```

or for CI scenarios these can be sourced from the environment with the format `<registryName>_USERNAME` and `<registryName_PASSWORD>`, e.g.

```
export MYOCIREGISTRY_USERNAME=spongebob
export MYOCIREGISTRY_PASSWORD=squarepants
```

If `<registryName>` contains hyphens, the environment variable to be read is the hyphen replaced by an underscore., e.g.

```yaml
repositories:
  - name: my-oci-registry
    url: myregistry.azurecr.io
    oci: true
```

```
export MY_OCI_REGISTRY_USERNAME=spongebob
export MY_OCI_REGISTRY_PASSWORD=squarepants
```

Attribution

We use:

- semtag for automated semver tagging. I greatly appreciate the author(pnikosis)'s effort on creating it and their kindness to share it!