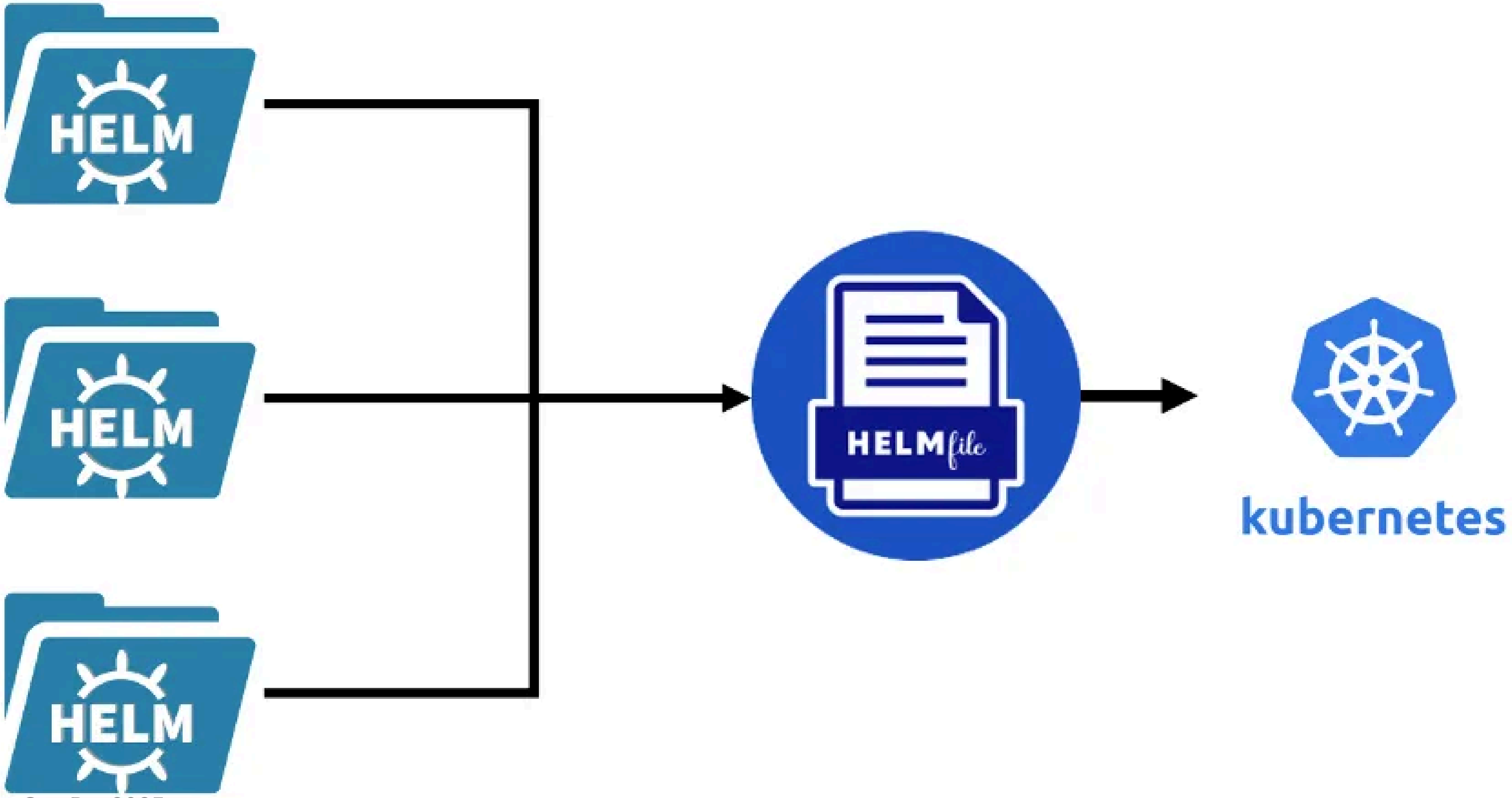
What is Helmfile?

- **Helmfile** is a declarative spec for deploying Helm charts.
- Manages multiple Helm charts as code.
- Simplifies complex Kubernetes deployments.


HELM
HELM
HELM
HELMfile
kubernetes

Why Use Helmfile?

- **Centralized Management**: Manage all Helm charts and environments in a single file.
- **Consistency**: Ensures repeatable, version-controlled deployments.
- **Simplicity**: Reduces manual steps and complexity.
- **Automation**: Integrates easily with CI/CD pipelines.
- **Environment Support**: Handles multiple environments without duplication.
- **Secret Management**: Securely manages sensitive data with tools like SOPS.

Key Features

- **Declarative YAML**: Define releases, values, and environments.
- **Environment Support**: Manage different configs for dev, staging, prod.
- **Secret Management**: Integrates with tools like SOPS for secrets.

Basic Structure of `helmfile.yaml`

The basic structure of a `helmfile.yaml` includes:

- **Releases**: Define the Helm charts to deploy, their names, and versions.
- **Environments**: Customize settings for different environments.
- **Values**: Specify custom configuration values for your charts.

Basic Structure of `helmfile.yaml`

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

releases:
    - name: my-app
        chart: stable/my-app
        namespace: default
        values:
            - common-values.yaml

    - name: my-database
        chart: stable/mysql
        namespace: default
        values:
            - common-values.yaml
            - database-values.yaml
```

Understanding the Example `helmfile.yaml`

Environments in Helmfile

- **Environments** allow you to define different configurations for each deployment scenario (e.g., staging, production).
- In the example:
  - **staging** uses `values-staging.yaml`
  - **production** uses `values-production.yaml`
- This separation keeps configurations organized and reduces duplication.

Releases in Helmfile

- **Releases** specify which Helm charts to deploy and how to configure them.
- Example defines two releases:
  - **my-app**: Deploys `stable/my-app` with settings from `common-values.yaml`.
  - **my-database**: Deploys `stable/mysql` with `common-values.yaml` and extra configs from `database-values.yaml`.

What Happens During Deployment?

- When you run `helmfile sync`:
  - Helmfile reads `helmfile.yaml`.
  - Applies the correct values for the chosen environment.
  - Deploys all defined charts as specified.

Labels in Helmfile

- **Labels** are tags you attach to your Helm releases.
- Help group and manage releases efficiently.
- Use labels to target specific releases when running Helmfile commands.

```
releases:
    - name: frontend
      chart: stable/frontend
      labels:
          env: production

    - name: backend
      chart: stable/backend
      labels:
          env: staging
```

Using Labels with `--selector`

- **Sync only production releases:**

```
helmfile --selector env=production sync
```

- **Sync only staging releases:**

```
helmfile --selector env=staging sync
```

Environments in Helmfile

- **Environments** let you define settings for different deployment scenarios (e.g., staging, production). Avoids duplication by keeping environment-specific configs in one file.

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

releases:
    - name: frontend
        chart: stable/frontend
        namespace: default
    - name: backend
        chart: stable/backend
        namespace: default
```

Deploying to Specific Environments

- **Deploy to staging:**

```
helmfile --environment staging sync
```

- **Deploy to production:**

```
helmfile --environment production sync
```

Environment Variables in Helmfile

- **Environment variables** allow dynamic values in your `helmfile.yaml`.
- Useful for injecting secrets or configuration at runtime.

```yaml
releases:
    - name: my-app
      chart: stable/my-app
      namespace: {{ requiredEnv "NAMESPACE" }}
      values:
          - replicas: {{ env "REPLICA_COUNT" | default "2" }}
```

Using Environment Variables

- **Set variables manually:**

```
export NAMESPACE=production
export REPLICA_COUNT=5
helmfile sync
```

- **Or load from a `.env` file:**

```
source .env
helmfile sync
```

Secrets Management in Helmfile

- Securely manage sensitive data (API keys, passwords) using tools like **SOPS**.
- Helmfile can decrypt secrets at deployment time.

```
sops --encrypt --output secrets.yaml <<EOF
db_password: my-secret-password
api_key: my-api-key
EOF
```

```
releases:
    - name: my-app
      chart: stable/my-app
      values:
          - values.yaml
          - secrets.yaml
```

Helm vs. Helmfile: Quick Comparison

Feature	Helm	Helmfile
Purpose	Package manager for Kubernetes charts	Declarative management of multiple Helm releases
Release Management	Individual releases	Multiple releases in one file
Environment Support	Limited	Multiple environments with custom values
Secrets Management	No built-in, relies on external tools	Integrates with SOPS or Kubernetes secrets
Use Case	Simple/individual charts	Complex/multi-chart, multi-env scenarios

Helmfile Best Practices Guide

A guide to advanced patterns and structuring for scalable, maintainable Helmfile usage.

.Values in Helmfile vs. Helm

- Both Helm and Helmfile use `.Values` in templates.
- In Helmfile, `.Values` refers to environment values, while in Helm it refers to chart values.
- Helmfile provides `.StateValues` as an alias for its own `.Values` to avoid confusion.

```
app:
    project: {{ .Environment.Name }}-{{ .StateValues.project }}
```

Handling Missing Keys and Defaults

- Helmfile fails if you reference a missing key in environment values.
- Use the `default` function to provide fallback values:

```
{{ .Values.eventApi.replicas | default 1 }}
```

- To allow missing keys without failure, use the `get` function:

```
{{ .Values | get "eventApi.replicas" nil }}
```

- Combine `get` and `default` for safe defaults:

```
{{ .Values | get "eventApi.replicas" 1 }}
```

Reducing Repetition: Release Templates

- Large projects often repeat fields like `namespace`, `chart`, `values`, and `secrets`.
- Use Helmfile's **Release Templates** to DRY up your configuration.

```yaml
templates:
    default:
        chart: stable/{{`{{ .Release.Name }}`}}
        namespace: kube-system
        missingFileHandler: Warn
        values:
            - config/{{`{{ .Release.Name }}`}}/values.yaml
            - config/{{`{{ .Release.Name }}`}}/{{`{{ .Environment.Name }}`}}.yaml
        secrets:
            - config/{{`{{ .Release.Name }}`}}/secrets.yaml
            - config/{{`{{ .Release.Name }}`}}/{{`{{ .Environment.Name }}`}}-secrets.yaml

releases:
    - name: kubernetes-dashboard
        version: 0.10.0
        inherit:
            - template: default
```

Release Template Features

- Templates support:
  - Basic fields: `name`, `namespace`, `chart`, `version`
  - Boolean fields: `installed`, `wait`, `verify`
  - Templated fields: `installedTemplate`, `waitTemplate`
  - `setTemplate`, `valuesTemplate`, and `secrets`
  - Inline values

```
setTemplate:
    - name: '{{`{{ .Release.Name }}`}}'
      values: '{{`{{ .Release.Namespace }}`}}'
```

Layering State Files

- Use **Layering** to share common configuration across multiple Helmfiles.
- Extract shared parts (like environments) into separate files.

```yaml
# helmfile.yaml
bases:
    - environments.yaml

releases:
    - name: metricbeat
      chart: stable/metricbeat
    - name: myapp
      chart: mychart

# environments.yaml
environments:
    development:
    production:
```

Merging Arrays in Layers

- Arrays (like `releases`) are **not merged** across layers.
- The last defined array overrides previous ones.

```yaml
# Layer 1
releases:
    - name: metricbeat
      chart: stable/metricbeat
---
# Layer 2
releases:
    - name: myapp
      chart: mychart
```

- Result: Only `myapp` remains.
- **Workaround:** Use YAML anchors or Go templates to import shared releases.

Layering State Template Files

- For even more DRYness, use **state template files** with Go templating.
- Each `---` -separated part is a template rendered in sequence.

```yaml
# helmfile.yaml.gotmpl
bases:
    - myenv.yaml
---
bases:
    - mydefaults.yaml.gotmpl
---
releases:
    - name: test1
        chart: mychart-{{ .Values.myname }}
        values:
            - replicaCount: 1
                image:
                    repository: "nginx"
                    tag: "latest"
```

Re-using Environment State in Sub-Helmfiles

- Load environment state once and pass it to sub-Helmfiles.

```yaml
environments:
    stage:
        values:
            - env/stage.yaml
    prod:
        values:
            - env/prod.yaml
---
helmfiles:
    - path: releases/myrelease/helmfile.yaml
        values:
            - {{ toYaml .Values | nindent 4 }}
```

Sub-Helmfile can use inherited values:

```yaml
releases:
    - name: mychart-{{ .Values.myrelease.myname }}
      installed: {{ .Values | get "myrelease.enabled" false }}
      chart: mychart
      version: {{ .Values.myrelease.version }}
      labels:
          chart: mychart
      values:
          - values.yaml.gotmpl
```

Summary

- Use `.StateValues` to distinguish Helmfile values.
- Handle missing keys with `default` and `get`.
- Reduce repetition with Release Templates.
- Layer state files for shared configuration.
- Be aware of array overriding in layers.
- Use state template files for advanced DRY patterns.
- Pass environment state to sub-Helmfiles for modularity.

Common Helmfile Commands

- **Template charts**

  ```
  helmfile template
  ```

- **Install or upgrade releases:**

  ```
  helmfile sync
  ```

- **Preview changes before applying:**

  ```
  helmfile diff
  ```

- **Apply changes with approval:**

  ```
  helmfile apply
  ```

- **Delete all releases:**

  ```
  helmfile destroy
  ```

Workflow Example

1. Write `helmfile.yaml` with releases.
2. Run `helmfile sync` to deploy all charts.
3. Use `helmfile diff` to preview changes.

Benefits

- **Consistency**: Same config for all environments.
- **Automation**: Integrates with CI/CD.
- **Scalability**: Manage many charts easily.

Conclusion

- Helmfile streamlines complex Kubernetes deployments.
- Ideal for teams managing multiple Helm charts and environments.

References

- What is Helmfile? - Devtron Blog
- Helmfile GitHub