Helm

Introduction to Helm

Introduction to Helm

Section overview

1 Helm and Kubernetes

1. Understand what Helm is and why it’s essential in Kubernetes.
2. Explain the benefits and limitations of using Helm to manage Kubernetes resources.

2 Helm vs. Kustomize

1. Understand benefits, limitations, and use-cases for both tools.

3 Helm Architecture

1. Explore the key components of Helm and how they interact.
2. Understand how Helm interacts with Kubernetes to manage releases.

Helm

Helm and Kubernetes

Helm and Kubernetes

Understand what Helm is and why it’s essential in Kubernetes.

Deploying applications with just Kubernetes poses many challenges:

Complexity in Resource Management

Multiple Resource Definitions and Dependencies: Deployments, Services, ConfigMaps, Secrets, etc. - managing everything in individual YAML files quickly gets overwhelming.

Error-Prone Manual Edits

Syntax and Configuration Errors: Manual YAML edits are prone to syntax issues, misconfigurations, or inconsistent object names.

Human Error: Manually ensuring each resource is configured correctly is challenging.

Environment Configuration Overload

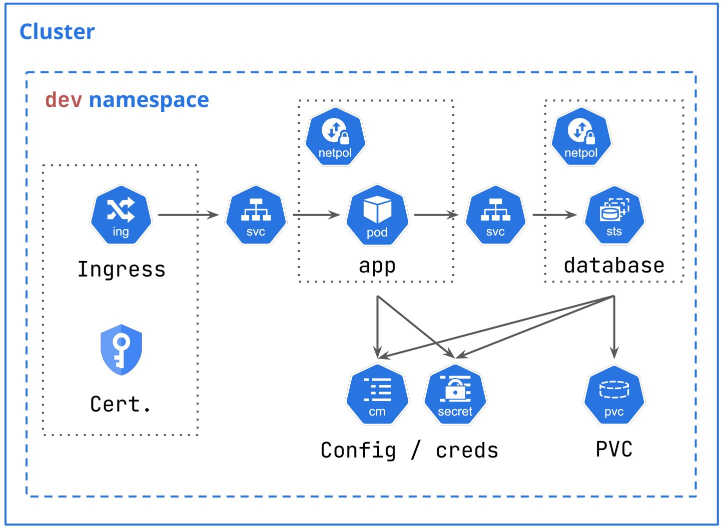
Duplication: Managing different configurations can lead to repetitive, nearly identical YAML files.

Limited Scaling of Kustomize: Lacks flexibility for complex, multi-environment deployments.

Version Control and Rollbacks

Tracking Changes: Maintaining version history for each YAML file is challenging and can result in configuration drift.

Rolling Back: In case of failure, there’s no straightforward way to revert to a previous state.

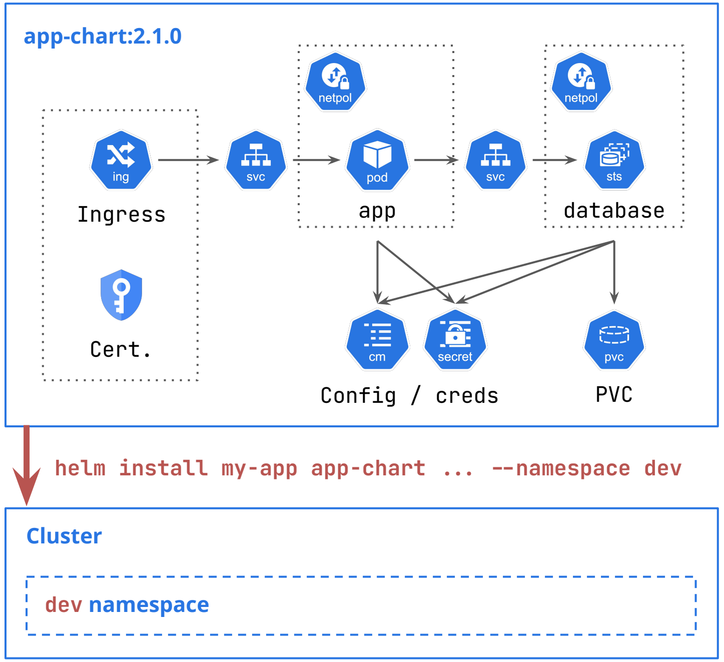


Helm

Helm and Kubernetes

Understand what Helm is and why it’s essential in Kubernetes.

- Helm is a **package manager** for Kubernetes, akin to apt for Debian, yum for RedHat, brew for MacOS, or npm for NodeJS.
  - It allows you to define, install, and upgrade applications containing multiple Kubernetes resources with a single command.
- With Helm, we can:
  - Package multiple Kubernetes manifests (called templates) into a single chart.
  - Deploy, update and manage all the manifests from a chart with a single command.
  - Thoroughly customize the templates with Go templating features and custom values files.
  - Create and leverage reusable charts available either publicly or privately.
  - Leverage charts for versioning your application.
  - Easily automate testing your charts with Helm hooks and the `helm` CLI.



Helm

Benefits and Limitations

Benefits and Limitations

Benefits and limitations of using Helm to manage Kubernetes resources

Benefits	Limitations
✅ **Simplified Deployment Process:** Easier to make sure all necessary Kubernetes components are installed, as well as to manage updates and patches across multi-component deployments.	⭕ **Learning Curve:** Helm introduces new concepts like templating and a specific chart structure, so it might take a while to get used to it. The Go templating language is also very rich and sometimes not straightforward.
✅ **Consistency Across Environments:** Ensure that deployments are consistent across your environments, while allowing environment-specific configuration via value overrides.	⭕ **Over-templating and Hard-to-Maintain Charts:** For simple applications, Helm might add unnecessary complexity. We might also run into the problem of over-engineering the templates, making them more complex and harder to maintain than necessary.
✅ **Efficient Release Management:** Perform upgrades and rollbacks of entire applications with single commands.	⭕ **Security Implications:** Using community charts requires careful review to avoid vulnerabilities.
✅ **Enhanced Collaboration:** Thousands of very useful charts are available in public repositories, and teams and companies can leverage private repositories to share and review charts, promoting consistency, standardization, and best practices. Helm also encourages documenting your Charts thoroughly, thus leading to easier-to-use reusable components.	⭕ **Release State is stored in the Cluster**, which means deleting it can lead to inconsistencies in Helm. Additionally, manually changing the deployed objects will lead to configuration drift between what Helm thinks is deployed and what is actually deployed.
✅ **Versioned Deployments:** Versioning charts goes a long way in ensuring stable versions of your packaged application.	⭕ **Upgrades might be challenging to perform**, and small mistakes in versions (for example, bumping only the minor version for something that should be considered a breaking change) can lead to big problems in upgrades.
✅ **Templating Flexibility:** Allows you to go beyond the limitations of Kubernetes and Kustomize to create truly flexible and configurable applications.	

Helm

Helm vs. Kustomize

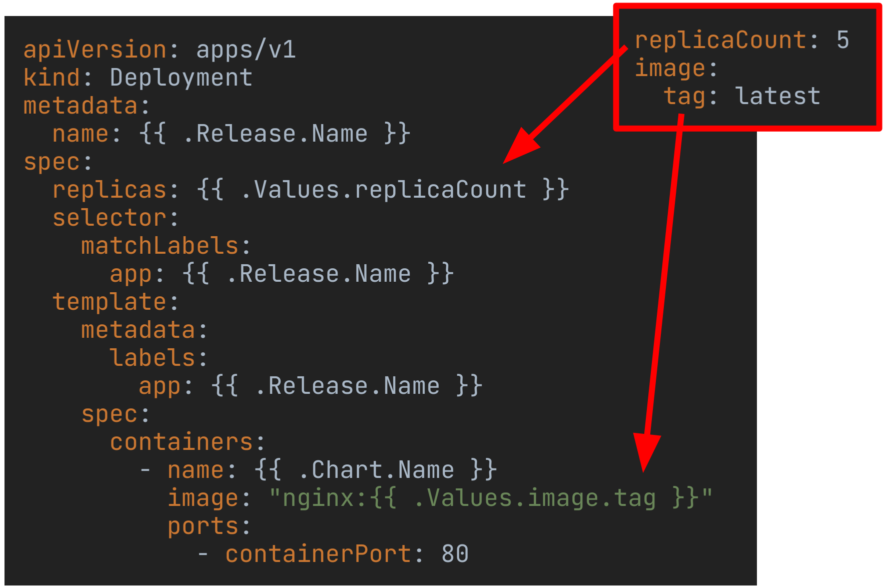
Helm vs. Kustomize

Understand the differences between the tools and their use-cases

Helm vs. Kustomize		
Dimension	**Helm**	**Kustomize**
Overall purpose	Package manager for Kubernetes with support for templating, dependency management, and versioning of applications.	Customize existing Kubernetes YAML manifests by overlaying changes also defined in YAML.
Complexity	More complex to work with, since it introduces the need to learn Go templates and the overall structure of charts.	Simpler to work with, since it leverages only native YAML constructs and does not introduce templating languages.
Customization features	Full templating system with conditionals, loops, functions, and variable substitution.	Strategic merge patches, JSON patches, name prefixes/suffixes, common labels, and annotations.
Use-cases	▪ Packaging and managing applications and their dependencies ▪ Versioning of applications ▪ More advanced customizations via templates and values files.	▪ Managing environment-specific customizations (e.g., dev, staging, prod) ▪ Applying patches and modifications without duplicating YAML

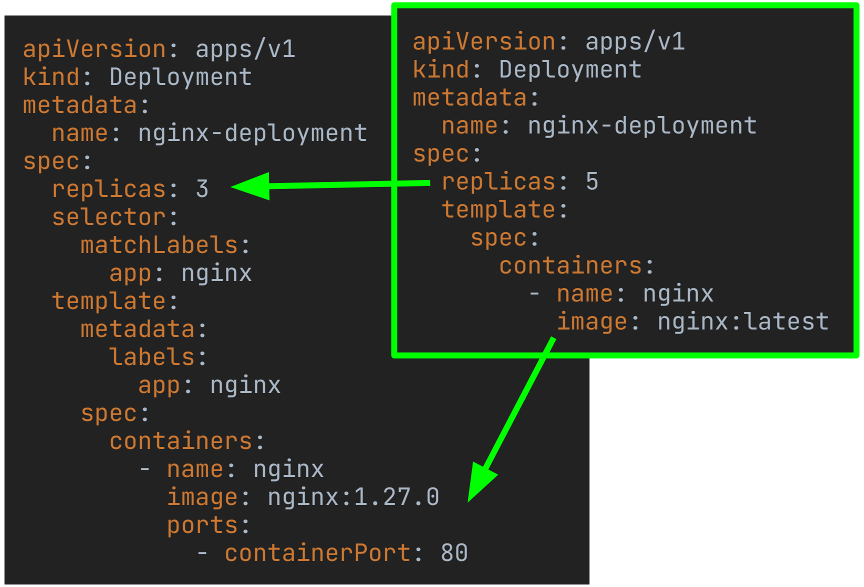
Helm vs. Kustomize

Understand the differences between the tools and their use-cases



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app: {{ .Release.Name }}
  template:
    metadata:
      labels:
        app: {{ .Release.Name }}
    spec:
      containers:
        - name: {{ .Chart.Name }}
          image: "nginx:{{ .Values.image.tag }}"
          ports:
            - containerPort: 80
```

```yaml
replicaCount: 5
image:
  tag: latest
```



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.27.0
          ports:
            - containerPort: 80
```

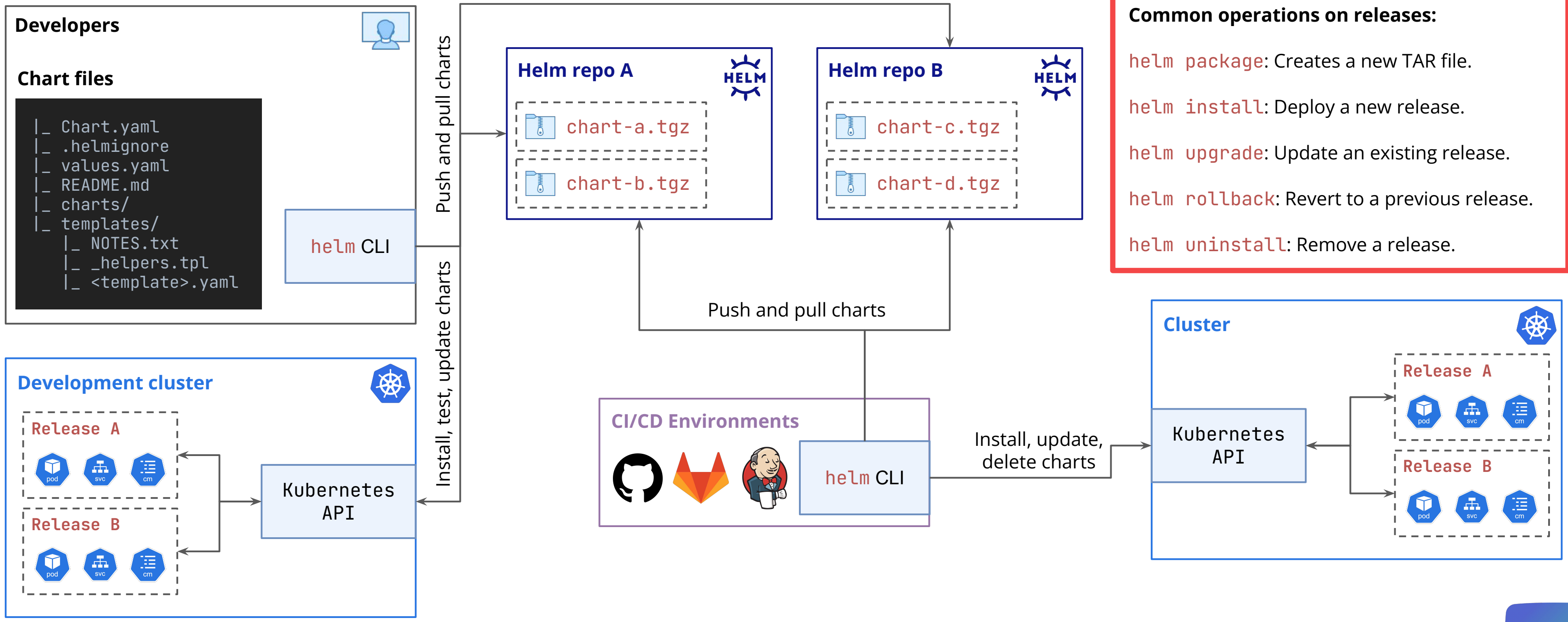
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 5
  template:
    spec:
      containers:
        - name: nginx
          image: nginx:latest
```

Helm

Helm Architecture

Helm Architecture

Understand the key components of Helm and how they interact



```
|_ Chart.yaml
|_ .helmignore
|_ values.yaml
|_ README.md
|_ charts/
|_ templates/
   |_ NOTES.txt
   |_ _helpers.tpl
   |_ <template>.yaml
```

Common operations on releases:

`helm package`: Creates a new TAR file.

`helm install`: Deploy a new release.

`helm upgrade`: Update an existing release.

`helm rollback`: Revert to a previous release.

`helm uninstall`: Remove a release.

Helm

Helm

Helm Fundamentals

Helm Fundamentals

Section overview

1 Explore Artifact Hub and Helm Repositories

1. Learn how to find and evaluate Helm charts using Artifact Hub.
2. Learn how to add, update, and manage Helm repositories.

2 Install and delete entire applications using Helm charts

1. Introduce the most common Helm CLI commands.
2. Visualize the results of installing applications via Helm.

3 Learn how to manage chart configuration values

1. Understand default configuration values.
2. Explore how to customize configuration via files.
3. Explore how to customize configuration via the CLI.

Helm Fundamentals

Section overview

4 Explore upgrading and rolling back Helm releases

1. Learn how to apply configuration changes to existing deployments.
2. Revert to previous versions of your applications using Helm’s rollback feature.
3. Understand how Helm manages release history and revisions.

Helm

Creating Helm Charts

Creating Helm Charts

Section overview

1 Understand the need for creating our own charts

1. Explore the importance and benefits of creating custom Helm charts.
2. Get an overview of the process involved in developing Helm charts.

2 Explore the standard chart structure and components

1. Understand the role and requirements of each file present in the chart.

3 Create our first Helm chart

1. Implement the chart's minimal structure.
2. Learn and practice the basics of templating with Go.
3. Learn how to package and publish our chart.

4 Explore how to use the `helm create` command to bootstrap charts

Helm

Why Creating Our Own Charts

Why Creating Our Own Charts

Explore the importance and benefits of creating custom Helm charts

Why build our own charts?

Off-the-shelf, well-maintained charts may not exist for your specific application.

Custom charts allow you to **define deployment configurations tailored to your app**.

Control over all Kubernetes resources and their configurations, with the possibility of implementing custom logic in the templates.

Versioning makes it easier to upgrade and rollback the many related objects of an application.

Enforce organizational policies and best practices, allowing teams to standardize how applications are deployed within your organization.

Reuse existing charts (either community or private) as dependencies for complex applications, while following best practices.

Helm

The Contents of a Helm Chart

The Contents of a Helm Chart

Understand the role and requirements of each file present in the chart

```
|_ Chart.yaml
|_ values.yaml
|_ README.md
|_ LICENSE
|_ .helmignore
|_ charts/
|_ templates/
   |_ deploy.yaml
   |_ svc.yaml
   |_ ingress.yaml
   |_ <others>.yaml
   |_ NOTES.txt
   |_ _helpers.tpl
```

The Contents of a Helm Chart

Understand the role and requirements of each file present in the chart

```
|_ Chart.yaml
|_ values.yaml
|_ README.md
|_ LICENSE
|_ .helmignore
|_ charts/
|_ templates/
   |_ deploy.yaml
   |_ svc.yaml
   |_ ingress.yaml
   |_ <others>.yaml
   |_ NOTES.txt
   |_ _helpers.tpl
```

`Chart.yaml`

A YAML file containing metadata about the chart. Among other fields, allows us to set:

- `apiVersion`: The chart API version (v1 or v2). For Helm 3, use v2.
- `name`: The name of the chart.
- `version`: The version of the chart (uses semantic versioning).
- `appVersion`: The version of the application enclosed (not Helm itself).
- `description`: A brief description of the chart.
- `type`: Type of chart (e.g., application or library).
- `keywords`: A list of keywords representative of the project.
- `dependencies`: A list of other charts that the current chart depends on.

`values.yaml`

A YAML file containing default configuration values for the chart. It's recommended to leverage the values.yaml file as much as possible to avoid hard-coding configuration into the chart.

The Contents of a Helm Chart

Understand the role and requirements of each file present in the chart

```
|_ Chart.yaml
|_ values.yaml
|_ README.md
|_ LICENSE
|_ .helmignore
|_ charts/
|_ templates/
   |_ deploy.yaml
   |_ svc.yaml
   |_ ingress.yaml
   |_ <others>.yaml
   |_ NOTES.txt
   |_ _helpers.tpl
```

`README.md`

Provides a human-readable documentation file that should contain:

- A high-level description of the application the chart provides.
- Prerequisites, requirements, and setup needed to run the chart.
- Descriptions of options in values.yaml and default values.
- Other relevant information for chart installation, configuration, or upgrade.

`LICENSE`

A plain text file containing the license for the chart (and chart applications, if relevant).

`.helmignore`

Used to ignore paths when packaging the chart (for example, local development files).

The Contents of a Helm Chart

Understand the role and requirements of each file present in the chart

```
|_ Chart.yaml
|_ values.yaml
|_ README.md
|_ LICENSE
|_ .helmignore
|_ charts/
|_ templates/
    |_ tests/
    |_ deploy.yaml
    |_ svc.yaml
    |_ <others>.yaml
    |_ NOTES.txt
    |_ _helpers.tpl
```

`charts/` directory

Contains any chart dependencies (subcharts). These dependencies should be informed in the `Chart.yaml` file, and will be downloaded and saved locally.

`templates/` directory

This directory contains multiple files that are relevant for Helm projects, including the multiple Kubernetes manifest templates that are rendered by Helm.

`tests/` directory

Contains tests to be executed when running the `helm test` command.

`NOTES.txt`

Its contents are printed on the screen upon successful chart installation or upgrade.

`_helpers.tpl`

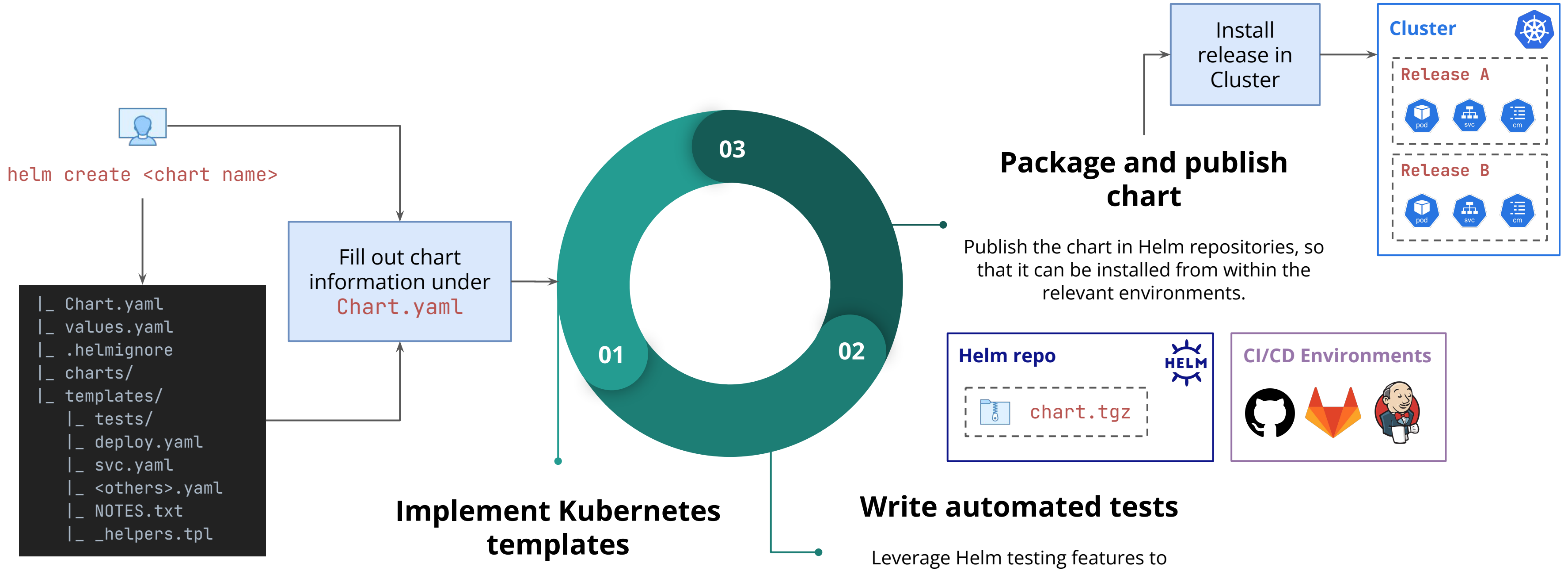
Contains template helper functions, which can be used to reduce duplication. Files preceded with an underscore are not included in the final rendering from Helm.

Helm

Recap: Creating Our Own Charts

Recap: Creating Our Own Charts

Recap the steps involved in developing and publishing Helm charts



```
|_ Chart.yaml
|_ values.yaml
|_ .helmignore
|_ charts/
|_ templates/
    |_ tests/
    |_ deploy.yaml
    |_ svc.yaml
    |_ <others>.yaml
    |_ NOTES.txt
    |_ _helpers.tpl
```

Package and publish chart

Publish the chart in Helm repositories, so that it can be installed from within the relevant environments.

Implement Kubernetes templates

Leverage configuration values, Go templating to create dynamic configuration, chart dependencies, among other features.

Write automated tests

Leverage Helm testing features to implement relevant tests to ensure Chart functionality.

Helm

Go Templating Deep-Dive

Go Templating Deep-Dive

Section overview

1 Take a deep dive into using functions and pipelines

1. Explore common templating functions and their syntax.
2. Understand how to compose pipelines containing multiple functions.

2 Learn how to create and reuse named templates

3 Explore control structures

1. Work with conditional logic within our templates.
2. Explore using loops to iterate through lists and objects.

4 Work with values, variables, scopes, and contexts

1. Explore how to set default values for optional configuration keys.
2. Practice how to define and use variables.

Go Templating Deep-Dive

Section overview

4 [Continued] Work with values, variables, scopes, and contexts

1. Understand how scopes and contexts work in Go templates.

5 Learn how to validate Chart values

1. Explore how to use required and fail functions.
2. Practice how to define reusable validation functions.

Helm

Managing Chart Dependencies

Managing Chart Dependencies

Section overview

1 Understand what subcharts are and how to manage them

1. Explore how to define chart dependencies (subcharts) in the `Chart.yaml` file.
2. Practice using the helm CLI to manage subcharts.

2 Learn how to pass values from parent charts to subcharts

3 Explore setting global values in parent charts

4 Learn how to include named templates from subcharts

5 Practice how to conditionally enable subcharts via both booleans and tags

What are Subcharts?

Understand the concept of chart dependencies and subcharts in Helm.

- A subchart, or a chart dependency, is another chart that is either:
  - Required for your chart to function properly.
  - Necessary to enable optional functionality in the installed application.
- Subcharts allow you to include and manage the deployment of other charts alongside your own.
- A few use cases for Subcharts include:
  - **Databases:** Including a database chart (e.g., MySQL, PostgreSQL) that your application depends on.
  - **Shared Services:** Including common services used across multiple applications.
  - **Common Utilities:** Including a library of functions or utilities to support Chart development
- Subcharts are placed under the `charts/` folder in your Helm chart. They can either be a folder containing all the required Chart files, or the tar file of an already existing chart.
  - Subcharts listed without a repository must contain all required files and be a valid Helm chart.
- Subcharts can be conditionally enabled either via boolean values or tags.

```yaml
apiVersion: v2
name: deps-demo
description: Illustrate deps
type: application
version: 0.1.0
appVersion: '1.16.0'

dependencies:
  - name: mongodb
    version: 16.2.1
    repository: <repo URL>
```

Managing Chart Dependencies

Explore the process for adding and updating chart dependencies.

```yaml
apiVersion: v2
name: deps-demo
description: Illustrate deps
type: application
version: 0.1.0
appVersion: '1.16.0'

dependencies:
  - name: mongodb
    version: 16.2.1
    repository: <repo URL>
```

`helm dependency list <chart dir>`

- Shows information about the installed dependencies for a specific chart.

`helm dependency update <chart dir>`

- Updates the `Chart.lock` file.
- Downloads and saves the dependencies `tar` files.

`helm dependency build <chart dir>`

- Downloads and saves the dependencies `tar` files.
- Fails if the informed version in the `Chart.yaml` is different from the `Chart.lock` file.

Helm

Advanced Topics

Advanced Topics

Section overview

1 Explore how to access files from within the Helm chart

1. Learn how to work with the `.Files` object and related functions.
2. Leverage glob patterns and `ConfigMap` and `Secret` utilities for better handling files.

2 Gain a thorough understanding of how to work with Helm hooks

1. Understand the different hooks available during Helm's release lifecycle.
2. Practice working with different hooks.
3. Learn how to set the hook execution order by using weights.
4. Explore how to handle hook failures, and common scenarios we might find ourselves in.

3 Explore creating library charts

1. Understand the difference between library and application charts.
2. Create and use our own library chart.

Advanced Topics

Section overview

4 Learn how to use Helm hooks for testing purposes

1. Work with the `test` hook and the associated Helm CLI commands.

5 Explore the Helm plugin ecosystem

1. Understand how plugins extend Helm's functionality.
2. Explore working with common plugins such as `helm-dashboard`, `helm-diff`, and `helm-unittest`.

Helm

Chart Hooks

Chart Hooks

Explore how to execute custom logic at certain points in a release lifecycle.

- Helm hooks are K8s resources with special annotations that designate them to be executed at particular points in a release lifecycle.
- They provide a way to intervene in the deployment process, allowing for tasks that need to occur before or after certain lifecycle events.

Available Hooks	
`pre-/post-install`	■ `pre-install`: Executes **after** templates are rendered, but **before any resources are created** in Kubernetes. ■ `post-install`: Executes **after all resources are loaded** into Kubernetes.
`pre-/post-delete`	■ `pre-delete`: Executes on a deletion request **before any resources are deleted** from Kubernetes. ■ `post-delete`: Executes on a deletion request **after all of the release's resources have been deleted**.
`pre-/post-upgrade`	■ `pre-upgrade`: Executes on an **upgrade request** after templates are rendered, but **before any resources are updated**. ■ `post-upgrade`: Executes on an upgrade request **after all resources have been upgraded**.
`pre-/post-rollback`	■ `pre-rollback`: Executes on a rollback request after templates are rendered, but **before any resources are rolled back**. ■ `post-rollback`: Executes on a rollback request **after all resources have been modified**.
`test`	■ Executes when the `helm test` subcommand is invoked.

Chart Hooks

Explore how to execute custom logic at certain points in a release lifecycle.

- Hooks are defined by adding special annotations to Kubernetes resources in traditional Helm templates:

```yaml
apiVersion: batch/v1
kind: Job
metadata:
 name: "{{ .Release.Name }}"
 labels:
   ...
 annotations:
   "helm.sh/hook": post-install,post-upgrade
   "helm.sh/hook-weight": "-1"
   "helm.sh/hook-delete-policy": hook-succeeded
```

`"helm.sh/hook": post-install,post-upgrade`

- Resources can implement one or multiple hooks.
- The configuration above will execute the hook both after a successful installation and a successful upgrade.

`"helm.sh/hook-weight": "-5"`

- Hook weights can be used to define the execution order of hooks.
- Weights are sorted in ascending order, which means lower numbers will be executed first

`"helm.sh/hook-delete-policy": hook-succeeded`

- This option determines when the corresponding hook resources will be deleted from Kubernetes.
- Possible values are: `before-hook-creation` (default), `hook-succeeded`, `hook-failed`.
- Similarly to the `helm.sh/hook` configuration, multiple values are allowed here.

Helm

Helm Plugins

Helm Plugins

Learn how to extend Helm's functionality by installing third-party plugins.

- Helm plugins provide additional functionality to Helm, but are not included in the base Helm binary.
- There are many useful plugins, and you can write your own if you wish to do so!

Popular Helm Plugins	
`helm-dashboard`	■ Offers a UI-driven way to view the installed Helm charts, see their revision history and corresponding k8s resources. ■ Allows users to perform simple actions such as rolling back to a revision or upgrading to a newer version.
`helm-diff`	■ Provides a preview of what the `helm upgrade` command would change. ■ It also supports comparing two revisions/versions of your helm release.
`helm-unittest`	■ Allows writing tests in YAML to assert template and chart characteristics. ■ Does not create any resources in the cluster.
`helm-secrets`	■ Allows encrypting sensitive information in the Helm charts and decrypting the values on the fly. ■ Also supports injecting secrets from external sources (for example, AWS SecretsManager) during a Helm deployment.