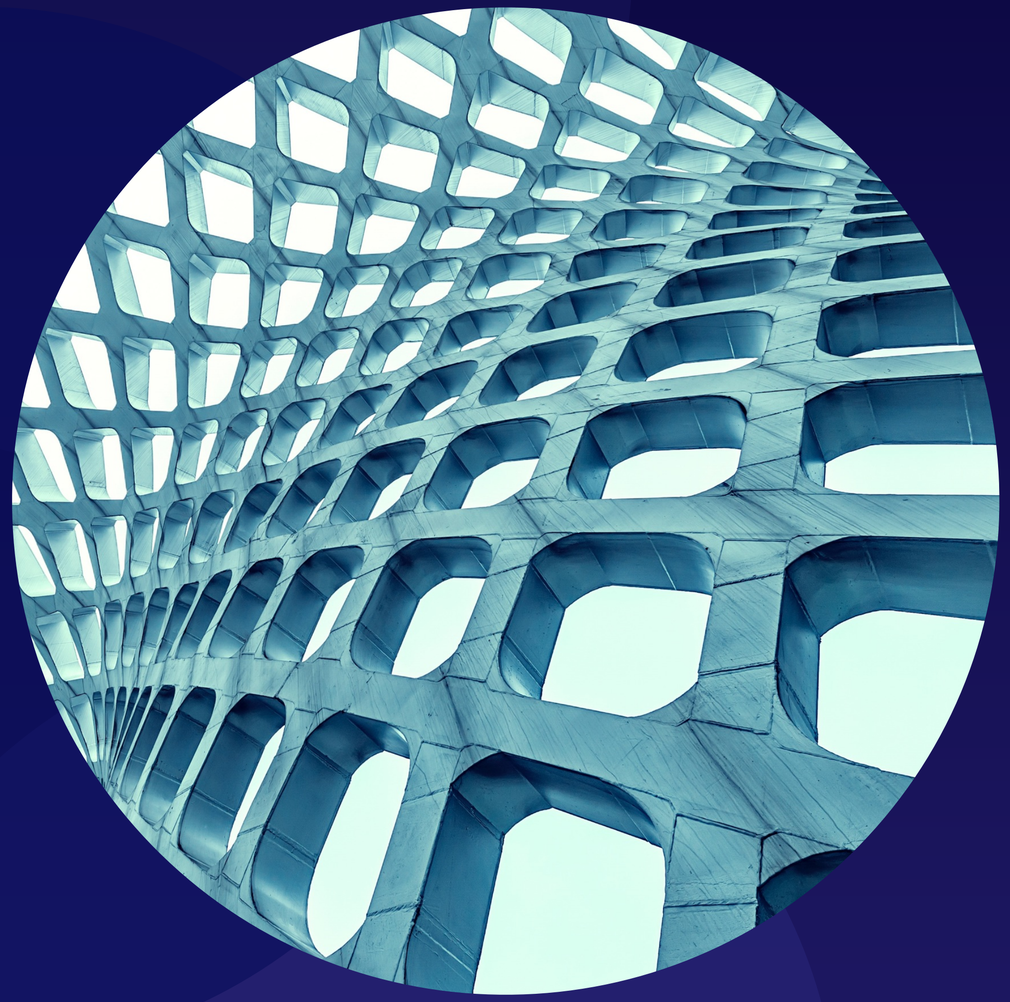
What Is Helm?



Helm is a package manager for Kubernetes.

What Is a Package Manager?

Managed

- Single command installation
- Dependencies are resolved
- Does not require deep understanding of software
- Easier updates, upgrades, and removal

VS

Unmanaged

- Dependencies must be satisfied manually
- Each part of the application must be installed, updated, upgraded, or removed separately
- Requires in-depth knowledge of the application architecture

How Packages Are Installed

Read Metadata

Packaged applications have accompanying data that indicates how it is installed (e.g., dependencies).

Installation

The pieces of the package are installed in order. This prevents failures by ensuring that requirements are met.

Resolve Dependencies

Not only do we know what the application stack depends on, we also know where to get it and how to install it.

Configuration

The installed components of a package may require post installation steps (e.g., linking databases).

Why Package Management Is Better

Old Manual Way

Someone with knowledge of the application would need to create an installation plan, create a dependency plan, and install and configure the application.

Package Management

Everything is in the package. Installation is typically one command. Once installed, the application is ready to use.

WHAT IS HELM?

How Helm Manages Packages

Helm Chart

This is the definition of a Kubernetes application. Helm uses this information along with a config to instantiate a released object.

Running vs Desired State

If a Helm chart has been released, Helm can determine what the current state of the environment is vs the desired state and make changes as needed.

Least Invasive Change

In the event that there is a change to a release, Helm will only change what has been updated since last release.

Release Tracking

Helm versions releases. This means that if something goes wrong, the release can be rolled back to a previous version.

What Helm Can Do for You

Single Command Install

With the `helm install` command, a chart can be released using a Helm repository.

Provide Insights for Releases

With the `helm status` command, it is possible to see the details of the running state of a release.

Perform Simple Updates/Upgrades

With the `helm upgrade` command, you can apply changes to a chart (e.g., versioning a service) and helm will do the update for you.

Provide the Ability to Rollback

Helm tracks releases and versions them. By using the `helm rollback` command, it is possible to revert to a previous release.

Simplify Deployment

Charts can be created by the application expert and released by someone else with a single command.

Single Command Uninstall

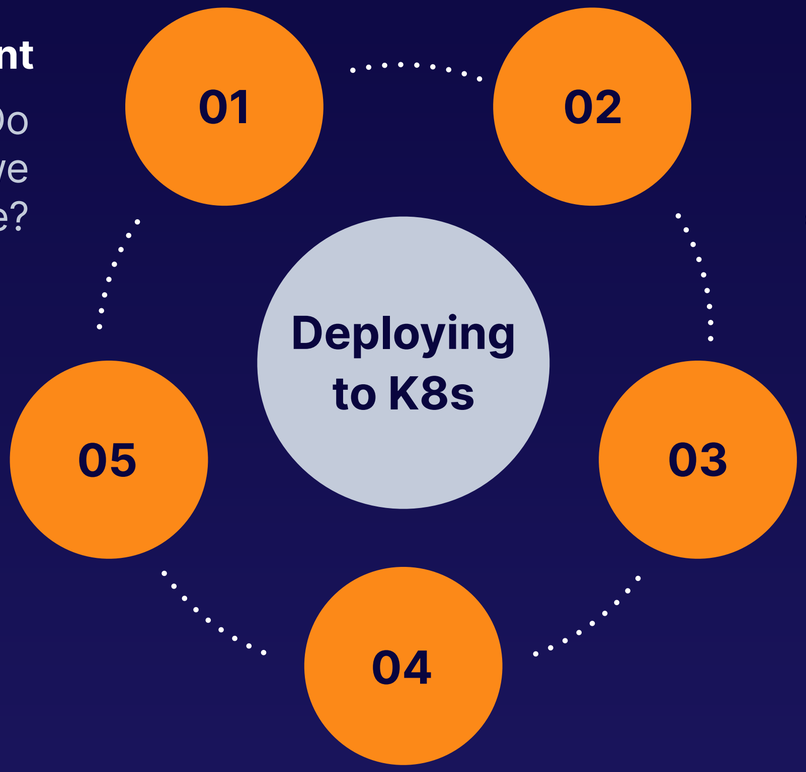
By using `helm uninstall`, the reverse of the installation can be done. This makes a cleaner removal, as all components that are defined are also removed.


A CLOUD GURU
Deploying to Kubernetes without Helm

Deployment Steps

Scope the Deployment

Determine what we need to create. Do we need persistent volumes? Do we need services for the database?



Create Dependencies

Create dependencies according to what we have determined is needed in the environment, such as persistent volumes.

Perform Configuration

Any application-specific configuration that needs to be done (e.g., logging in to the application) can now be done.

Create Manifests

We will need to create manifests for each of the objects that we want to deploy (e.g., if we need a database, we need to handle the login info).

Deploy Manifests

The manifests that we have created need to be deployed in the correct order. We need to ensure that the deployment is successful before moving to the next item.


A CLOUD GURU
Deploying to Kubernetes Using Helm

A CLOUD GURU

Deployment Steps

1

Locate a Chart

For the majority of applications you may want to deploy, there is an existing chart. These charts are located in a repository and can be deployed using some custom configurations.

2

Deploy the Chart

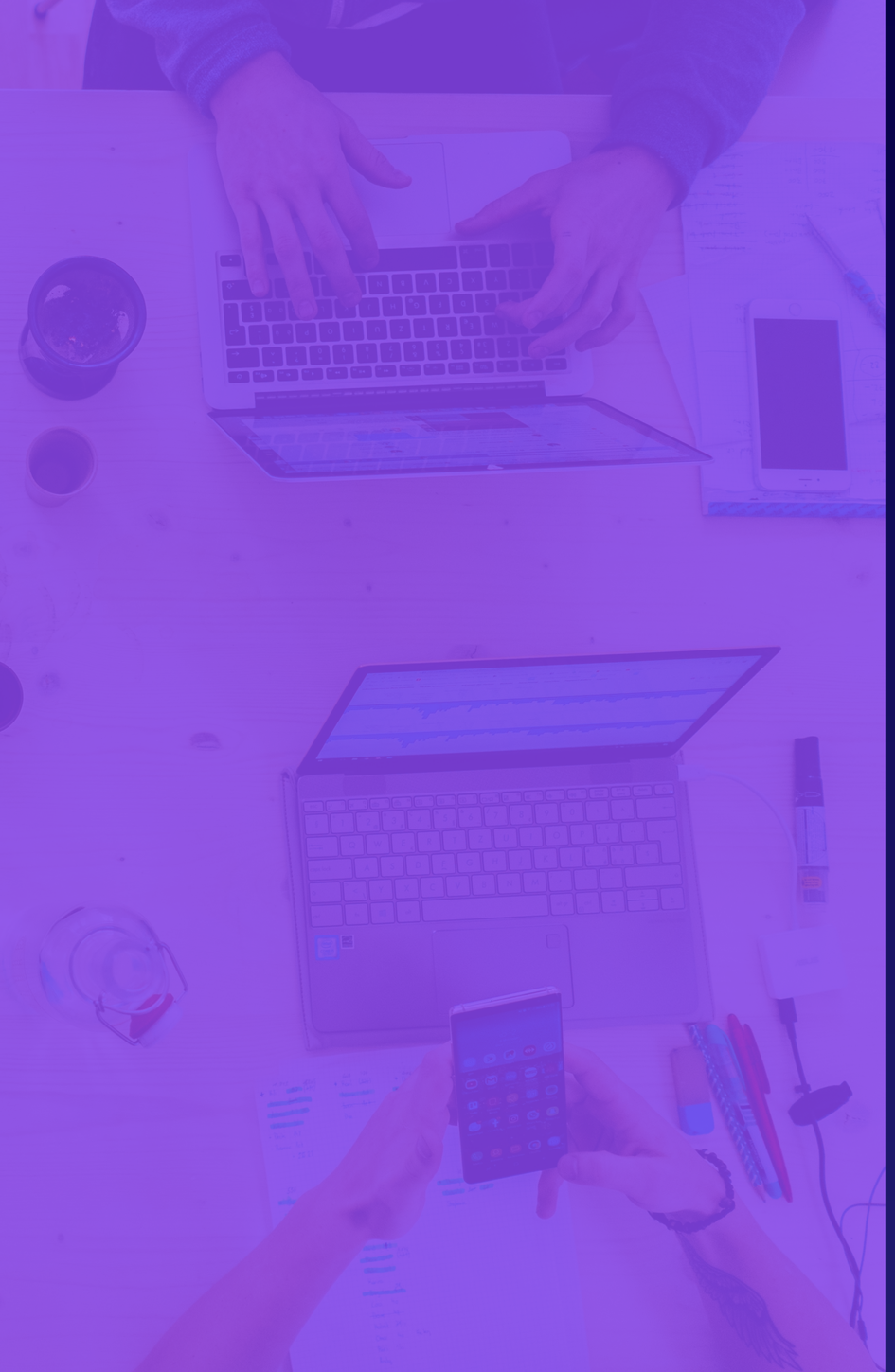
With Helm, all parts of the application package can be deployed using a single command.

3

Perform Configuration

Some applications you deploy may require post-installation configuration.

Installing Helm



A CLOUD GURU

Before We Begin

You will need to have a Kubernetes cluster installed and configured so that `kubectl` is working correctly. If you plan to install a chart that uses persistent storage, you need to have your storage classes configured correctly so the PVCs can be created.

A CLOUD GURU

Installation Methods

Package Manager

The Helm community has made packages available for Homebrew, Chocolatey, and APT, as well as a Snap package.

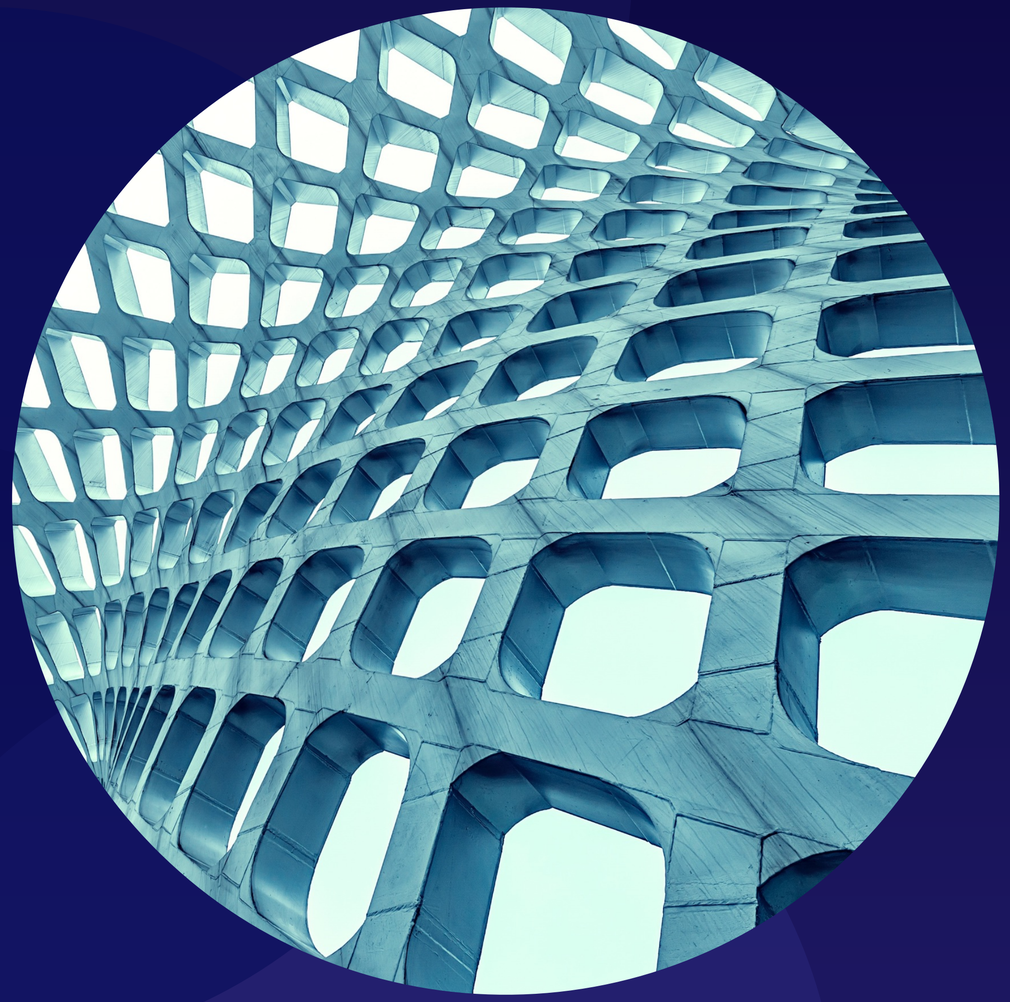
Provided Script

Helm has a provided script that will install Helm locally in bash.

Manually

The Helm binary can also be used to install Helm.

Post Installation Configuration

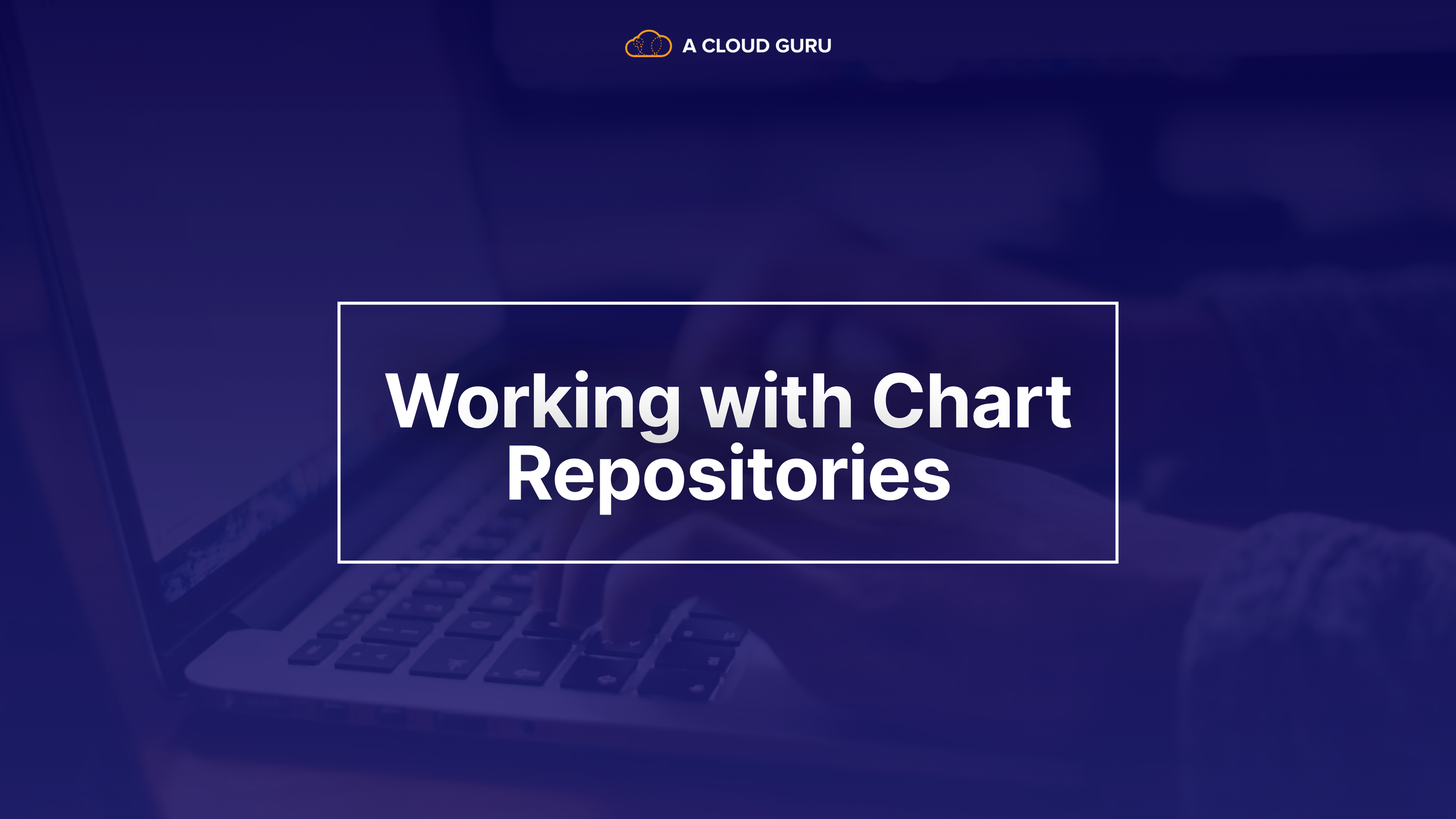


The Next Steps

Once you have Helm installed, you will need to add repositories so Helm is able to download charts. I recommend you also update the repository once you have it added.

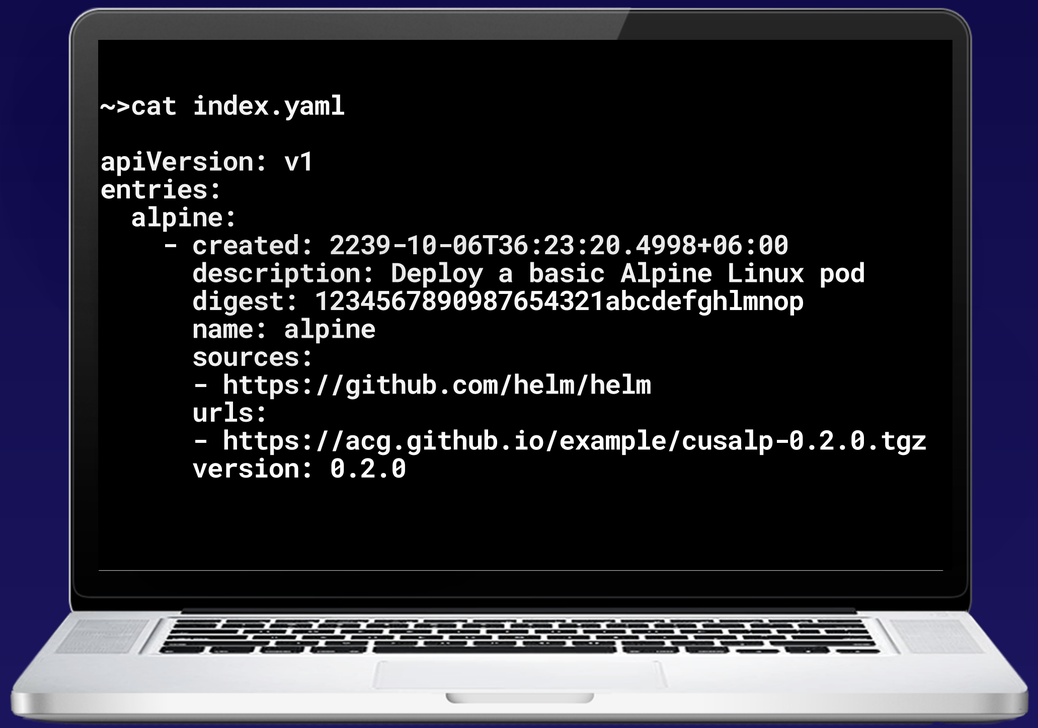
Ensure the Helm search function is working correctly.

Additionally, you might want to do a quick demo install to ensure Helm is able to install and delete a release correctly.



Working with Chart Repositories

What Is a Repository?



A chart repository is an HTTP server that is capable of serving an index.yaml file. This file contains the manifests of packaged charts and where they are located.

The index file for the repository can be created by using the `helm repo index` command and providing the directory which contains the packaged charts.

A packaged chart is a Helm chart that has been processed into a tar archive by Helm, typically using the `helm package` command and providing the chart name. These archives have a .tgz extension.

Useful Commands

A repository can be added to your Helm installation using the `helm repo add` command and providing the URL of the repository.

A list of the current repositories can be displayed using the `helm repo list` command. This shows locally cached repositories.

Helm caches repository information locally and at times it needs to be updated. This can be done using the command `helm repo update` .

Repositories can be removed from Helm using the `helm repo remove` command and providing the URL of the repository.

Updating Releases in Helm

Releases in Helm

A chart contains the scaffold of objects in Kubernetes without specific versions or values.

When a chart is installed, the scaffold is combined with the values to instantiate objects in the Kubernetes cluster. This combination of chart and values is called a **release.**

Values are the version and other information that a chart requires.

Releases in Helm

Releases in Helm

RELEASE

```
Name          version     image     version
Demo-release     1        alpine     1.0.6
Demo-release     1        mysql      3.06.2
```

The `helm status` command can be used to see the current state of a release. Helm keeps track of all the releases that were created with Helm and adds a version number for each named release.

```
helm upgrade Demo-release -set alpine.image=1.1.6
```

RELEASE

```
Name          version     image     version
Demo-release     2        alpine     1.1.6
Demo-release     2        mysql      3.06.2
```

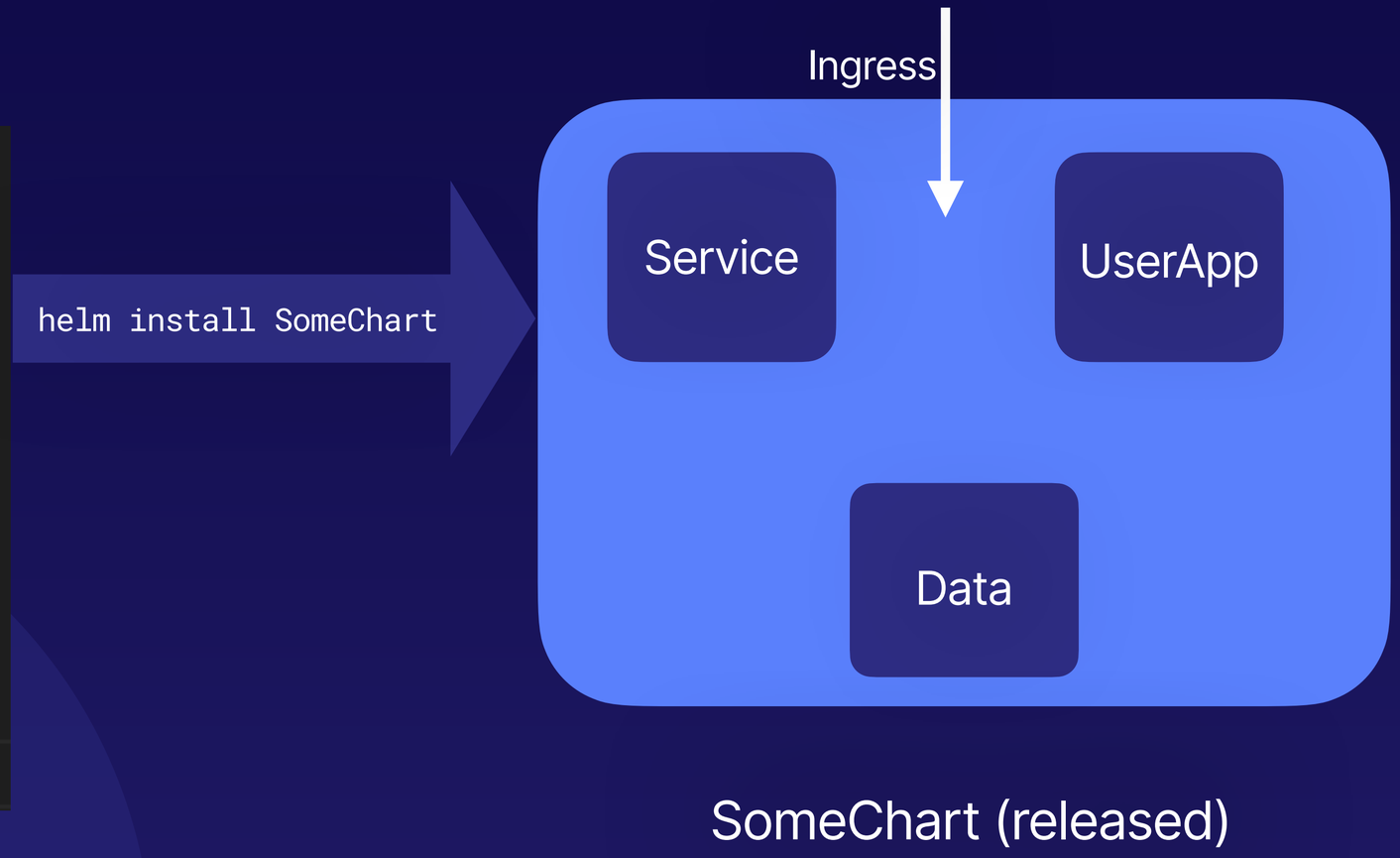
The `helm upgrade` command can be used to update a release. This can be done by specifying an updated chart or by modifying the release. Helm will only update what has been changed and increment the version.

Getting Into Helm Charts

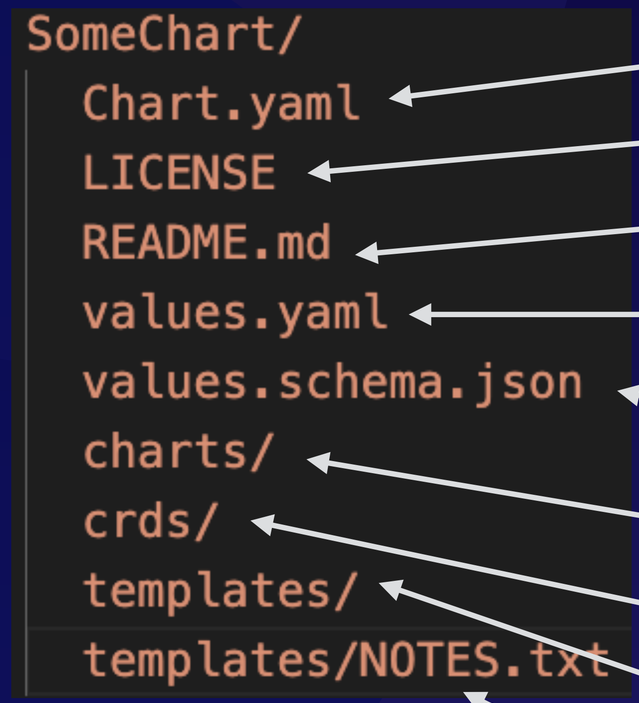
Structure of a Chart

A Helm chart is set of files that represents Kubernetes resources.

```
SomeChart/
  Chart.yaml
  LICENSE
  README.md
  values.yaml
  values.schema.json
  charts/
  crds/
  templates/
  templates/NOTES.txt
```



Structure of a Chart



Contains information about the chart, dependencies, version, etc.

Contains license information, if any.

Markdown-formatted chart description, options, etc.

Contains predefined (default) values which will be passed to the templates. Values in this file can be overridden by values passed on the command line during installation.

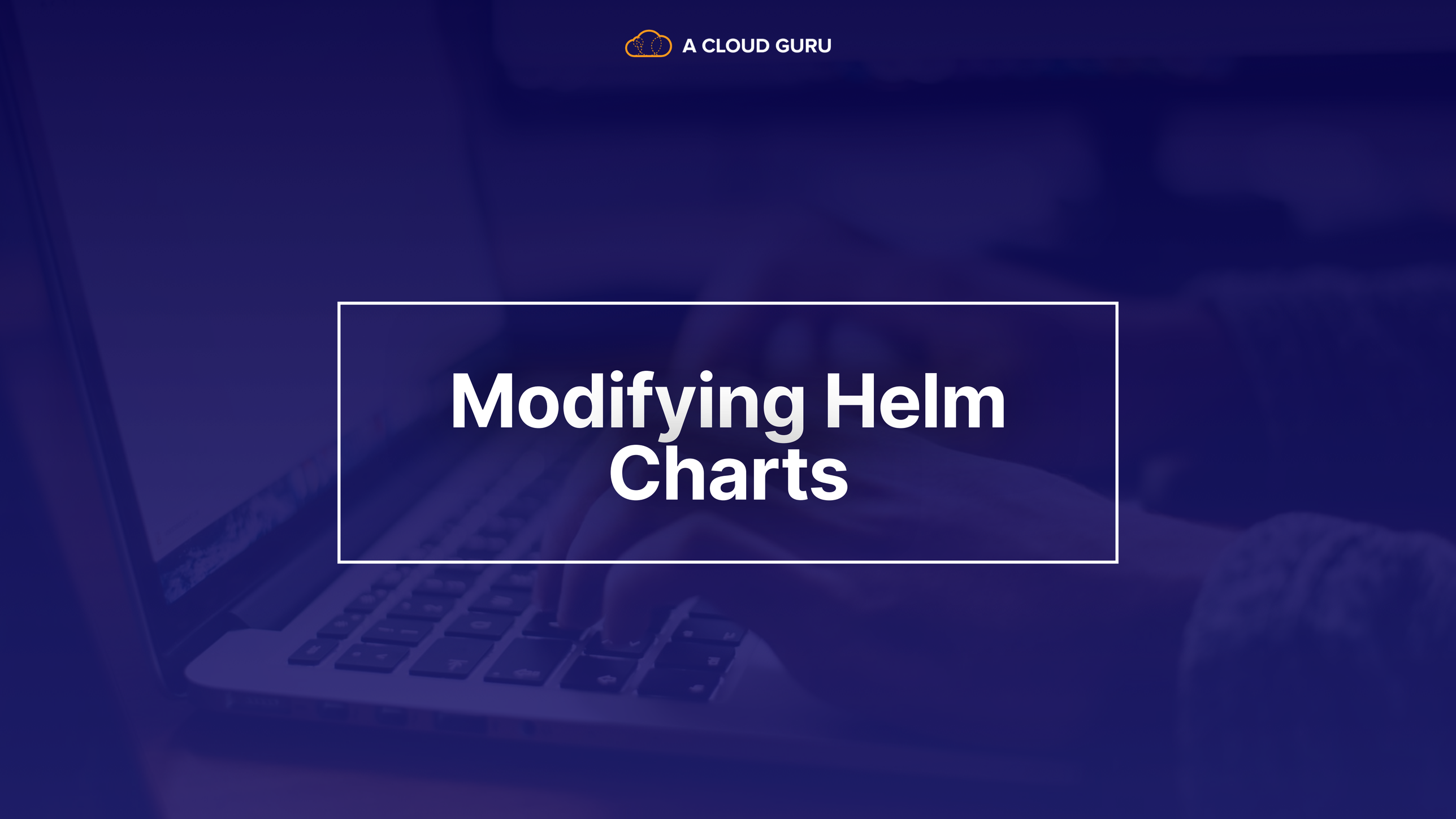
Provides an optional schema for the `values.yaml` file.

Dependencies are downloaded into this folder.

Contains custom resource definitions (CRDs).

This directory contains templates which are passed through the template engine. These templates incorporate the contents of the `values.yaml` file.

This plain text file is printed out post-installation and when viewing release status.



Modifying Helm Charts

Modifying Helm Charts

There are several ways to modify a Helm chart.

***Pro tip:**

The command `helm show values` can be used to show all of the information contained in the `values.yaml` file.

Inline using set

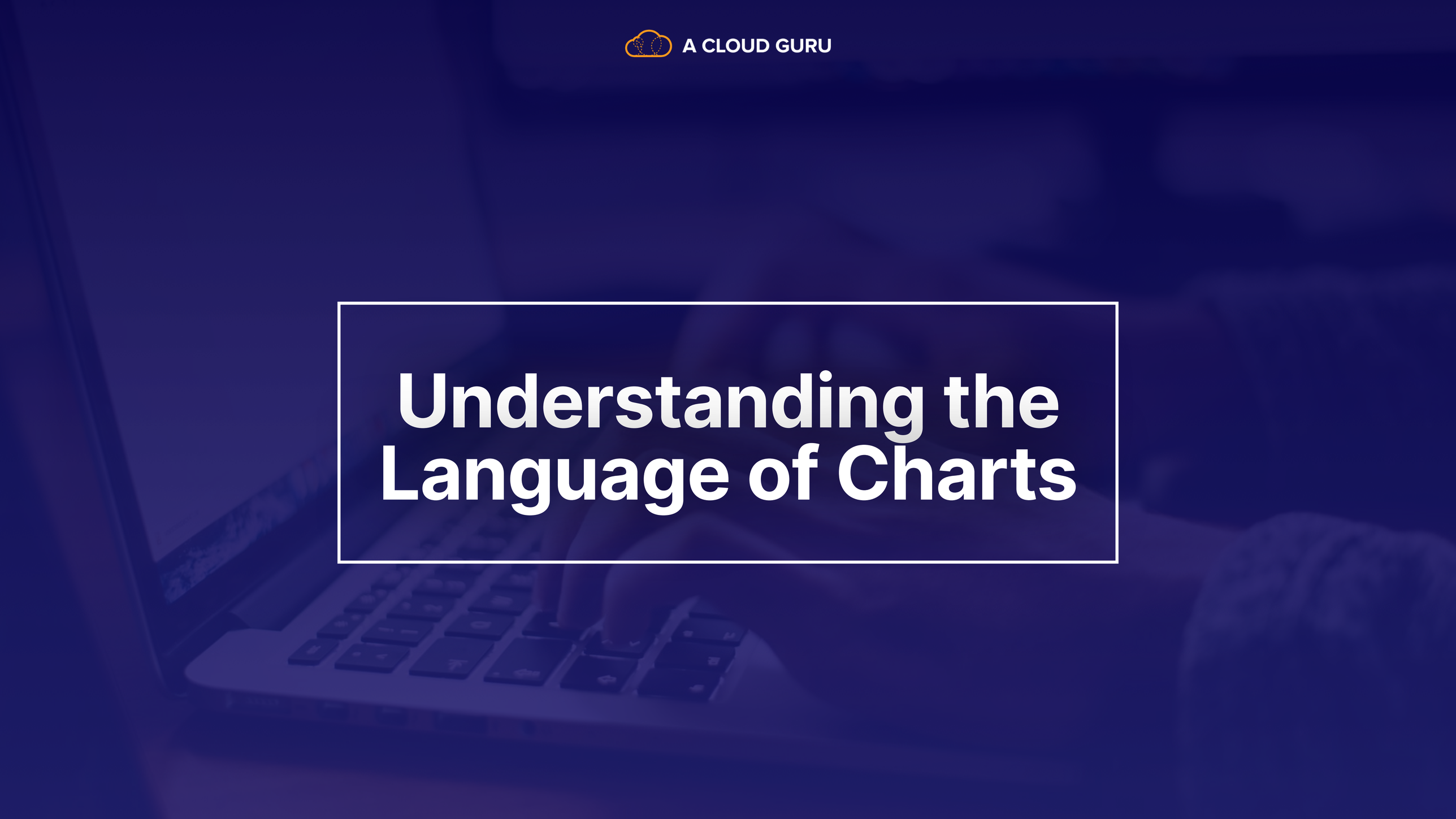
If you are modifying a few values and are testing your overrides, you can use the set option with the install command to override the contents of the `values.yaml` file.

Passing a file

You can override the chart's `values.yaml` by passing a file that contains the values you want to override. This has the advantage of being able to commit the override file to source control to track changes.

Customize the Chart

You can "fetch" the chart and then directly overwrite the `values.yaml` with your custom values. This is a destructive change and creates a new version of the chart.



Understanding the Language of Charts

The Benefits of Templates

Reusability

Templating allows a write once, use anywhere methodology. If there is a value that is not implementable, it can be overridden.

Pluggable

Sections of a template that are known to work in one chart can be used in another chart and the values file updated.

Versatility

With no hardcoded versions and no hard requirements, a template can be used with any valid values file.

Maintainability

Well-defined values allow only the values file to be the focus of updates. If a version or replica number needs to be updated, it only needs to be updated in one place.

Helm Templating Language

Based on the Go templating language.

Directives are denoted by double curly braces (e.g., `{{ object }}`). The object name is a combination of its values as shown below. In this example, the object is the registry we will use for the container. In the template, it would look like this:

```
1  ##chartname: somechart
2  image:
3    registry: docker.io
4    repository: example/somechart
5    tag: 5.5.3-debian-10-r11
6    pullPolicy: IfNotPresent
7  htaccessPersistenceEnabled: false
8  customHTAccessCM:
9  replicaCount: 1
10 extraEnv: []
11 extraVolumeMounts: []
12 extraVolumes: []
13 resources:
14   limits: {}
15   requests:
16     memory: 512Mi
17     cpu: 300m
```

`{{ .Values.image.registry }}`

This would result in the value in the chart `docker.io`.

Top Level

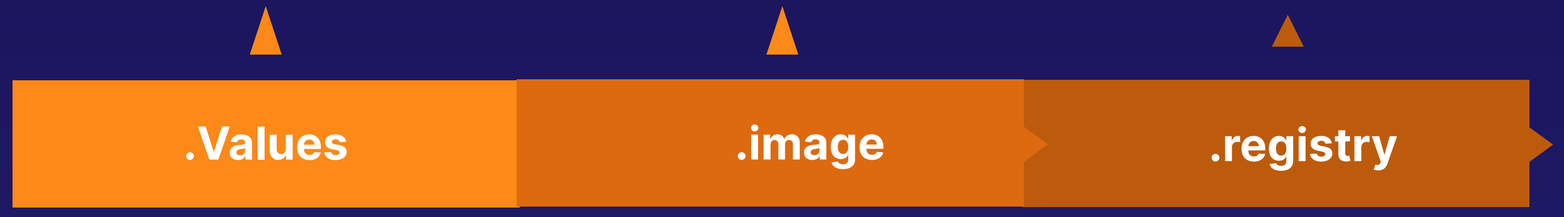
The “ . ” at the start indicates the top level and is typically the chart directory, so this would find values starting at the chart folder.

Next level

This is the next level from the top and is separated with a “ . ”.

Nested level

If the value is nested deeper, there can be another level also separated by a “ . ”.



Processing Charts

Charts can invoke conditionals and methods.

```
#values.yaml
securityContext:
  enabled: true
  runAsUser: 1001
#deployment.yaml
  {{- if .Values.securityContext.enabled }}
      securityContext:
        runAsUser: {{ .Values.securityContext.runAsUser }}
```

Here we want to use the values only if the security context is set to enabled.

We first perform the check using an if statement.
Then, we substitute the value.

Charts can use pipelining much like in Linux.

Here, we reference a value and pass the returned value to a pipe.

On the other side of the pipe is a `quote` function. This works like a print statement.

```
#values.yaml
mappingValues:
  firstOne: 'valueone'
  secondOne: 'valuetwo'
#deployment.yaml
data:
  primary: {{ .Values.firstOne | quote }}
  secondary: {{ .Values.secondOne | quote }}
```

Speaking the Language of Charts

Creation Process


```
SomeChart/
  Chart.yaml
  LICENSE
  README.md
  values.yaml
  values.schema.json
  charts/
  crds/
  templates/
  templates/NOTES.txt
```

Creating a Chart from Scratch

1. **Determine what you are releasing.**

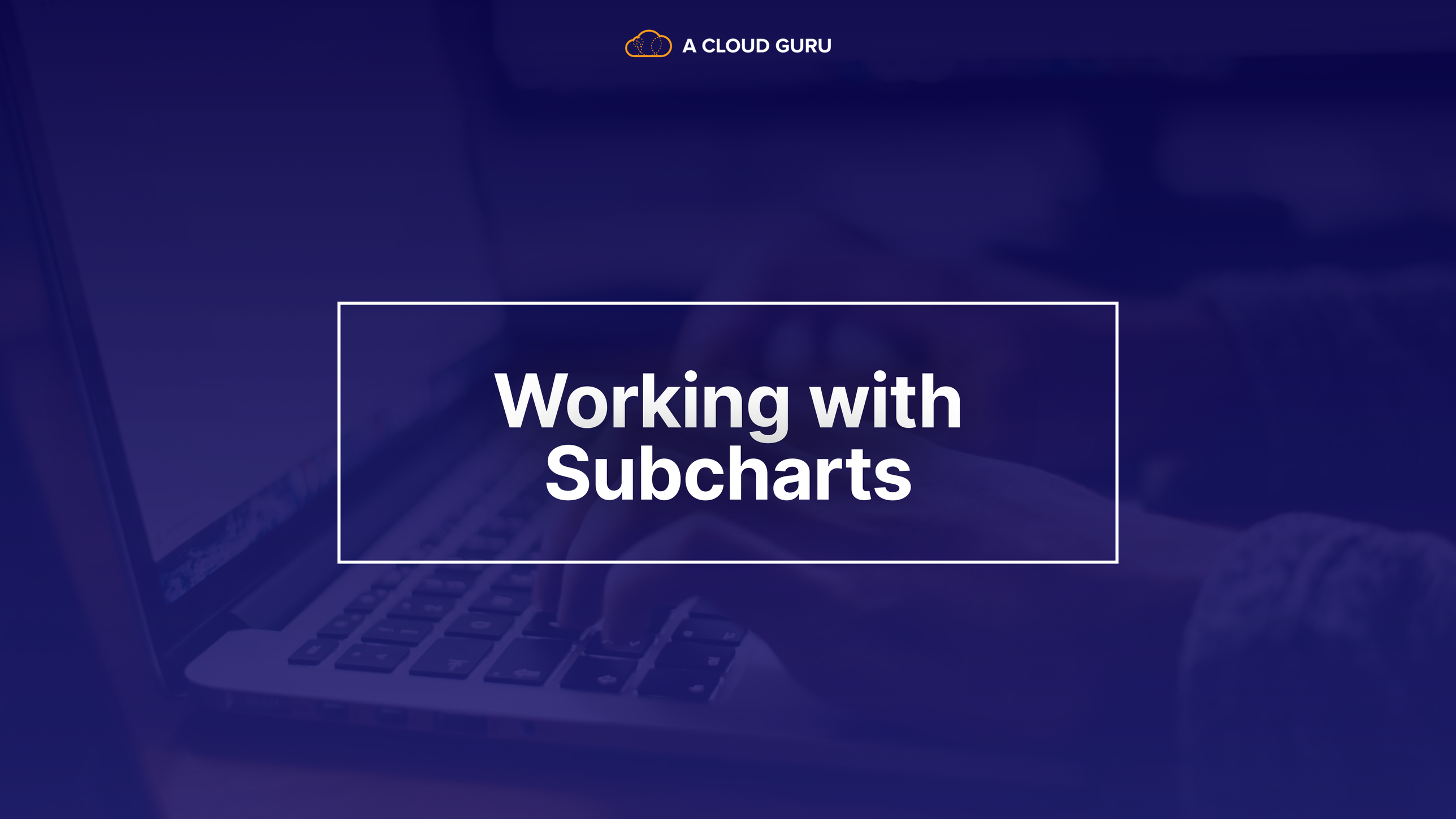
   Scope your project so that you know what you need to create.

2. **Create static manifests.**

   Create manifests for your release to ensure that what you are releasing works.

3. **Convert to templates and values.**

   Take the working manifest and convert it into a Helm chart, create a values file, and insert directives to consume the values and configs.



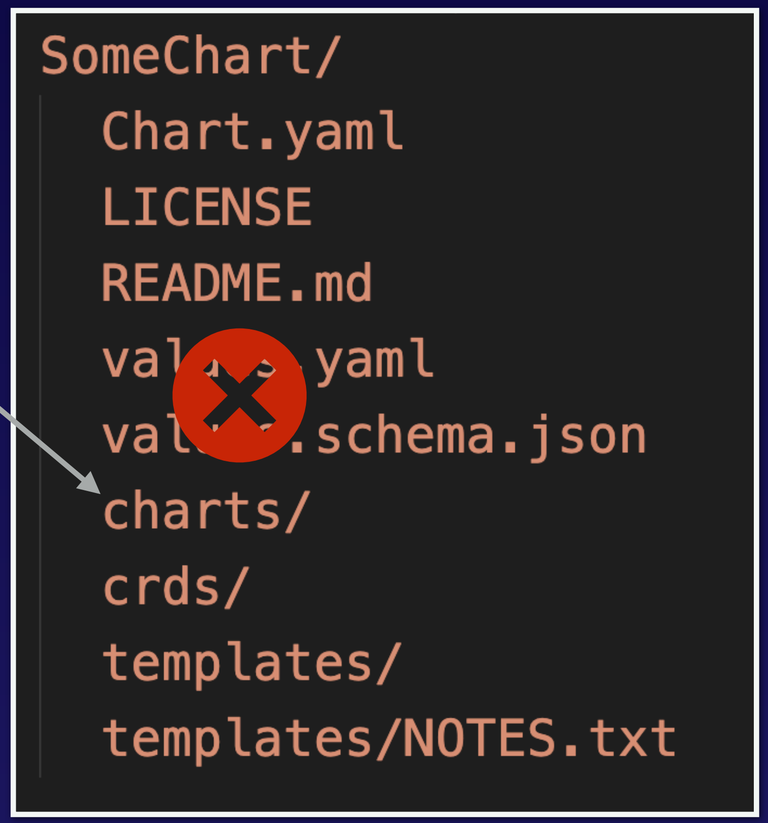
Working with Subcharts

Subcharts

A subchart is a child of its parent and is located in the `charts` directory of the parent chart.

A subchart is a standalone chart that cannot access the values of its parent.

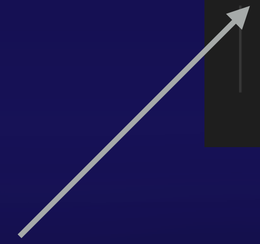
The parent chart can explicitly override the values of the subchart by defining the value in its values file.



Parent Override


```yaml
#values.yaml
mappingValues:
  firstOne: 'valueone'
  secondOne: 'valuetwo'

subchartValue:
  somesubchartvalue: 'override'
```



The parent chart can explicitly override the values of the subchart by defining the value in its values file.

Global Values


```yaml
#values.yaml
mappingValues:
  firstOne: 'valueone'
  secondOne: 'valuetwo'

subchartValue:
  somesubchartvalue: 'override'

global:
  thisvalue: 'something'
```

Global values can be accessed by both the parent and any subcharts that are present. They are defined using the global keyword.

This refers to the parent chart values → `somechart.mappingvalues.firstOne`

This refers to the subchart values → `somechart.subchartValue.somesubchartvalue`

This refers to the global values → `somechart.global.thisValue`

Implementing Pre- and Post- Actions with Hooks

Chart hooks provide a mechanism to exert control over chart processing.

Load Data

A hook can be used to load data such as secrets during the charts lifecycle to ensure that it is available.

Manipulate Data

Prior to an upgrade, a chart hook can be used to backup a database instance and restore it post-upgrade.

Stage Environments

Chart hooks can be used to execute jobs before, during, and after a chart runs, allowing services and other Kubernetes objects to be manipulated at that time.

Hook Types

Pre-Install

Executes after templates are rendered but before objects are instantiated.

Post-Install

Executes after all objects have been instantiated.

Pre-Delete

Executes on a deletion request before any objects are deleted.

Post-Delete

Executes after all of the released objects have been removed.

Pre-Upgrade

Executes on an upgrade request before any objects are updated.

Post-Upgrade

Executes after the upgrade request has been processed completely.

Pre-Rollback

Executes on a rollback request after templates are rendered but before any objects are modified.

Post-Rollback

Executes at the completion of a rollback operation after all objects are modified.

Important Hook Information

1 Annotations Define a Hook

A hook is a normal template that has some special annotations that define it as a hook.

2 Hooks Are Part of a Chart

Hooks are created in the templates section of a chart and belong to that chart or subchart. A parent chart cannot override the subchart's hooks.

3 Hooks Have Weights

Hooks can be ordered by weighting, meaning that the hook with the lowest weight executes first. If weighting is not important, it is best practice to set it to 0.

4 Hooks Fail Everything

If a hook is a job or a pod and it fails to complete or reach the ready state, then the release that contains the hook will fail. This is also true for a post- hook.

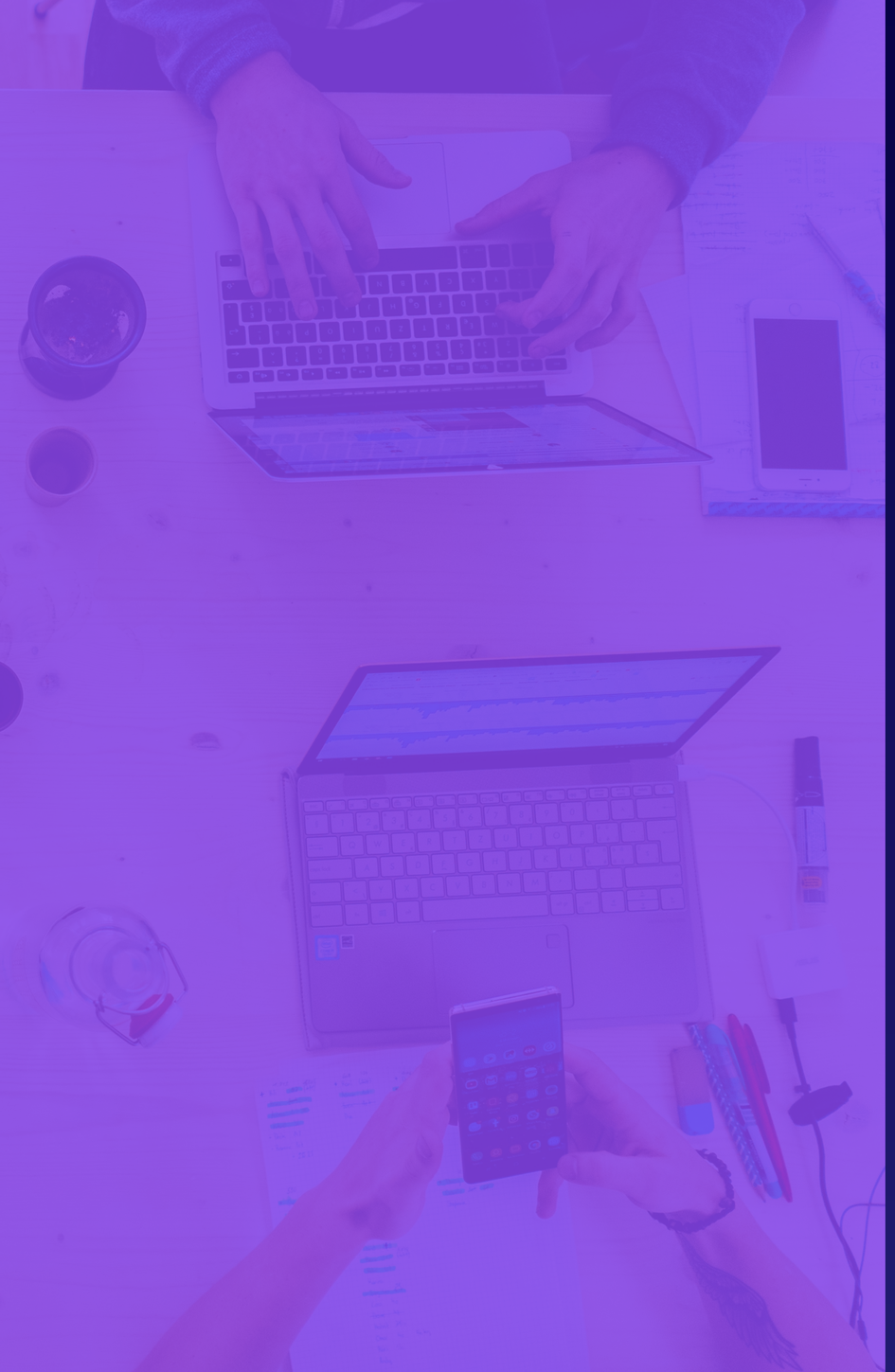
5 Multiple Hooks Can Be Defined

There is no limit to the number of hooks that can be defined in a chart. There can be both pre- and post- hooks defined in the same hook template.

6 Hook Objects Stand Alone

Objects created by hooks are not a part of a release and must have some consideration for tear down and clean up. When the release is deleted, the hook resources can persist.

Testing Charts



A CLOUD GURU

A Test Is a Special Type of Hook

A chart test is a job definition that creates a container with a running command to assert a condition. If the command succeeds, then the container should exit with a success code.

A CLOUD GURU

Testing Goals

Password Validation

Ensure that the username and password were created correctly.

Service Validation

Make sure that services are running and a service is available.

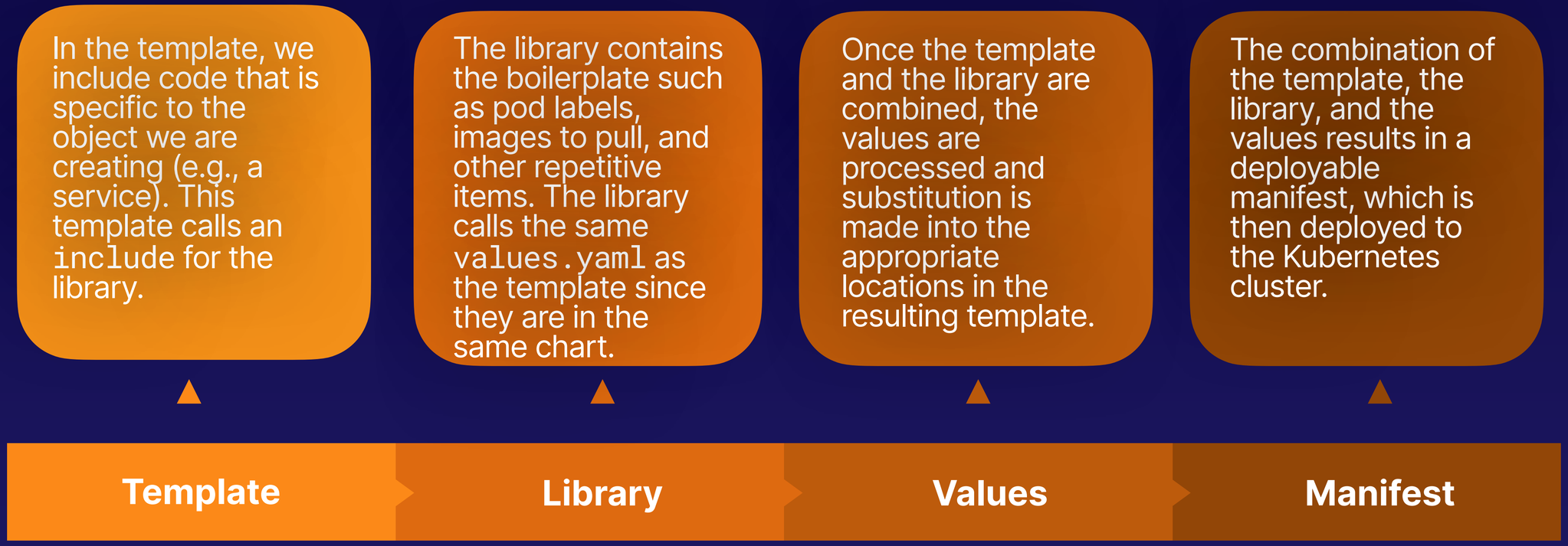
Configuration Validation

Make sure that load balancing is working correctly and ports are set correctly.

Creating and Using Libraries

What Is a Library?

A *library* is a shared template that can be used to prevent repetitive code.



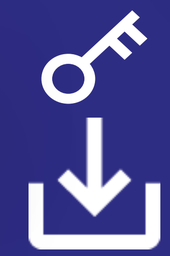
Validating Charts

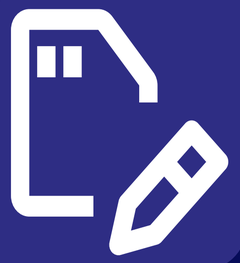
Provenance and Integrity

Packaging and Validating Helm Charts

`helm package -sign`

A key is used to generate a hash of the chart





This produces the helm chart in .tgz format

A provenance file is also generated and should be stored with the chart in its repo

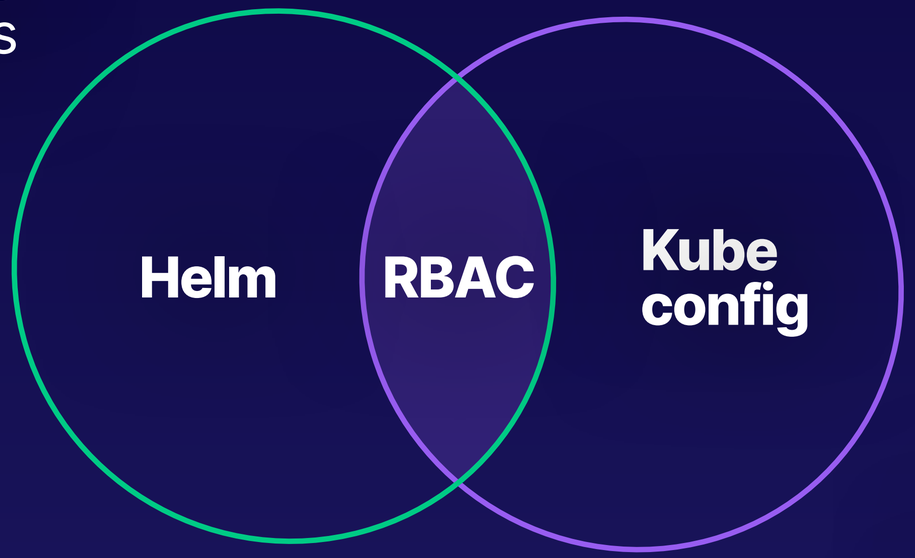
`helm verify`

The `verify` command can be used to validate the chart using the public key

Adding Role Based Access Control

Kube config

Helm uses the user's context to execute `kubectl` commands when it needs to interact with the Kubernetes cluster.



The config file determines the context the user is running in, which means the same permissions apply to helm as they would to a user invoking `kubectl`.

Troubleshooting Helm

Failure Modes

Helm Binary

Issues with the helm binary include permissions, versions, and access. The helm binary needs to be able to interact with kubectl, and this configuration should also be explored.

Chart Configs

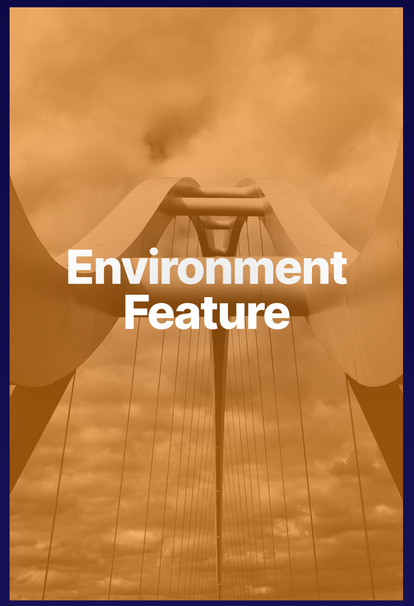
Is your chart written correctly or do you have syntax errors? Looking at the output of the error, is it a chart error or a manifest error? This is the place where - -dry-run is your best tool.

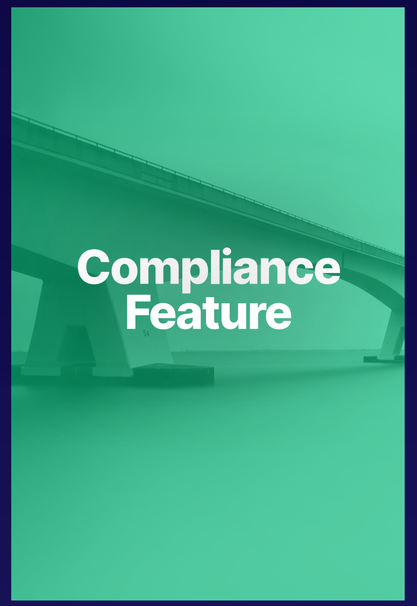
Kubernetes

Does your Kubernetes cluster work correctly? Are you able to perform commands directly using kubectl from the location where helm is installed?

Working With Plugins

Why Use Plugins?





There are some cases in which the basic functionality of Helm is not capable of meeting specific requirements for your environment. In cases like this, it may be necessary to use a plugin to provide that functionality. Plugins are not required to be programmed in Go and they are not required to be added to the core of Helm.

About Plugins


```
$  helm plugin install
```

This command can point to a url (similar to *wget* or *curl*) or it can be directed at a version control repository.

```
HELM_PLUGINS=
```

This directory (which is accessible using the *helm env* command) is the location where plugins are installed.

```
$HELM_PLUGINS/
  |- plugin-name/

       |
       |   - plugin.yaml
       |   - script.sh
```

A plugin is like a chart in that it has a structure starting with a top-level directory and a plugin.yaml file

The script in this example is optional, but a plugin executes a command that is listed in the plugin yaml. There is also a 'platform' command which is used first if present.

Exploring The Storage Backend

A CLOUD GURU

3 Types of Storage

1

ConfigMap

This stores release information in configmaps. This means that with RBAC configured, it might be less secure, as this data might contain passwords and other sensitive data.

2

Secret

In Helm v3, this is the default storage backend, and it is located in the Helm namespace.

3

SQL

For large deployments you can use a SQL database as the backend. This requires a deployed SQL instance and a configured connection string.

* at this time only PostgreSQL is available

Summary

A CLOUD GURU

Package Management

1

Automated Installation

Intimate knowledge of the software that is being installed is not necessary. Packaged applications can be installed with standardized commands.

2

Version Management

Version tracking in the package management system can be used to perform updates. Helm has the advantage of only replacing components that have been updated as necessary.

3

Automated Removal

Using standard commands, the packages can be removed and cleanup can be performed in an automated manner. This is especially useful in test systems.

A CLOUD GURU

Helm Manages Packages in Kubernetes

Helm

Helm is a client tool that runs alongside kubectl and uses kubectl to install packaged applications into a Kubernetes cluster.

Charts

Helm applications are packaged into charts. A chart is a group of files that define resources in Kubernetes.

Repository

Helm charts are stored and sourced from repositories. There are standard and custom repositories.

Helm Charts

Packaged as a TAR File

A chart is a set of files and directories that describe Kubernetes resources. When packaged, they become a TAR file that is located in the repository.

Can Be Modified

Charts can either be downloaded and extracted to be modified, or modifications can be done inline using the `set` command to override values in the chart.

Processed into Manifests

Charts are processed into manifests that are then installed via kubectl. The manifests can be seen by using the dry-run switch when running helm install.

A CLOUD GURU

Helm Administration

Security

Helm uses kubectl to compose and deploy manifests. This means that RBAC rules that secure your kubectl commands will also secure the execution of helm commands.

Configuration

Helm uses the kubectl configuration for its interface with the Kubernetes cluster. Helm itself has very little configuration, with the exception of the storage backend and plugins.

Troubleshooting

Because helm uses kubectl to interface with the Kubernetes cluster, most release issues can be determined by verifying that kubectl is working on the cluster.