

TEMPLATING YAML IN KUBERNETES

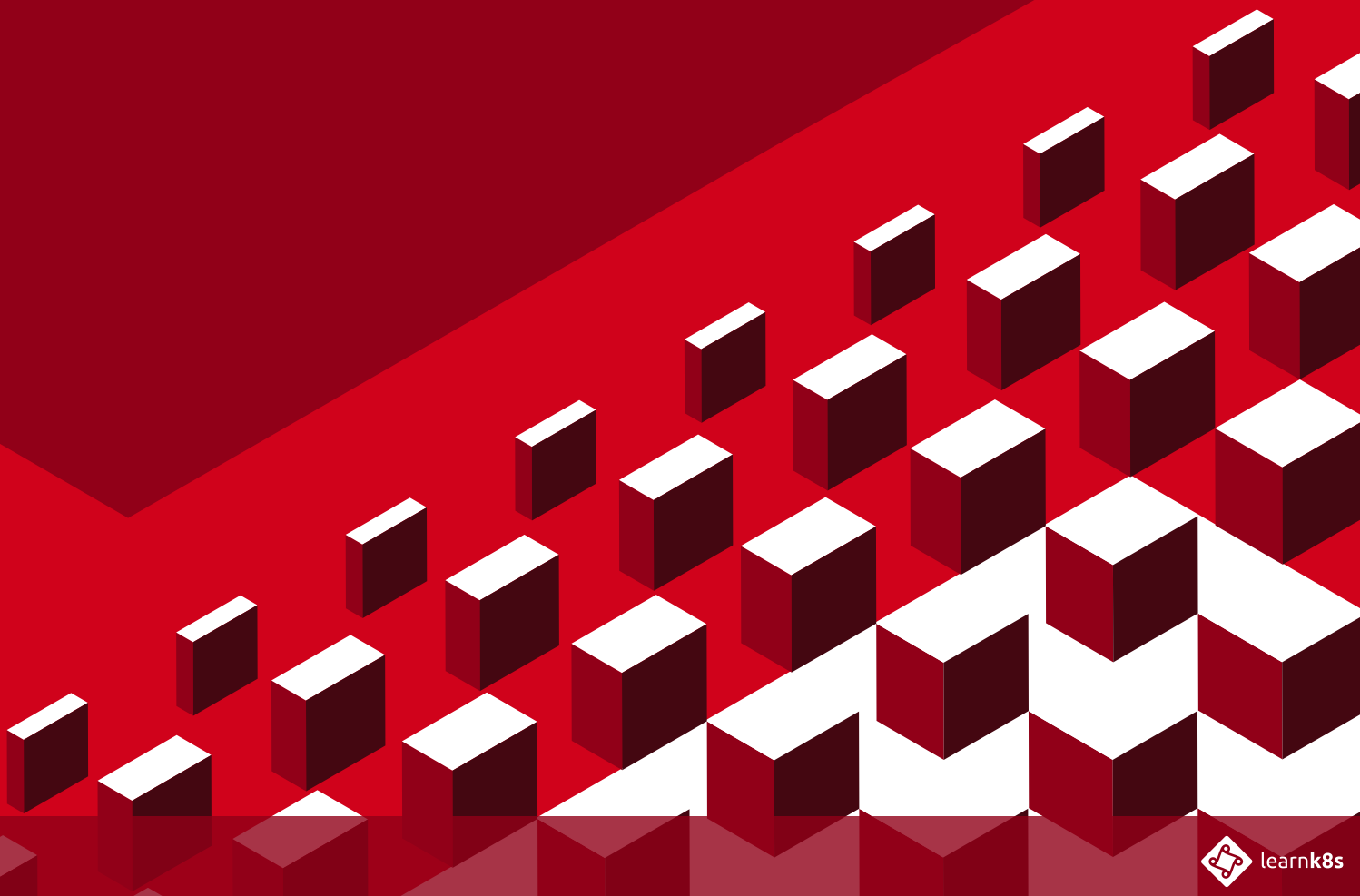


Table of contents

| | |
|---|-----------|
| 1. Templating YAML resources | 4 |
| Using templates with search and replace | 6 |
| Helm — the templating engine | 9 |
| Charts: Helm's packages | 10 |
| Helm — the release manager | 13 |
| Organising snippet of code into templates | 16 |
| Managing releases and rollbacks in Helm | 17 |
| Discovering and sharing charts | 21 |
| Declaring charts as dependencies | 21 |
| Using private registries to share charts | 22 |
| Helm 2 architecture | 23 |
| Helm 3 architecture | 26 |
| 2. Installing Helm | 27 |
| Prerequisites | 29 |
| 3. Creating charts | 30 |
| Deploying the chart | 31 |

| | |
|--------------------------------|-----------|
| Testing the application | 34 |
| <hr/> | |
| 4. Templating resources | 36 |
| <hr/> | |
| Templated deployments | 38 |
| <hr/> | |
| Defining your variables | 41 |
| <hr/> | |
| Overwriting custom values | 42 |
| <hr/> | |
| Dry-run | 43 |
| <hr/> | |
| Managing releases | 44 |
| <hr/> | |
| 5. Helpers | 46 |
| <hr/> | |
| Partials | 47 |
| <hr/> | |
| 6. Updating charts | 51 |
| <hr/> | |
| Upgrading releases | 52 |
| <hr/> | |
| Rolling back releases | 53 |
| <hr/> | |

Chapter 1

Templating YAML resources

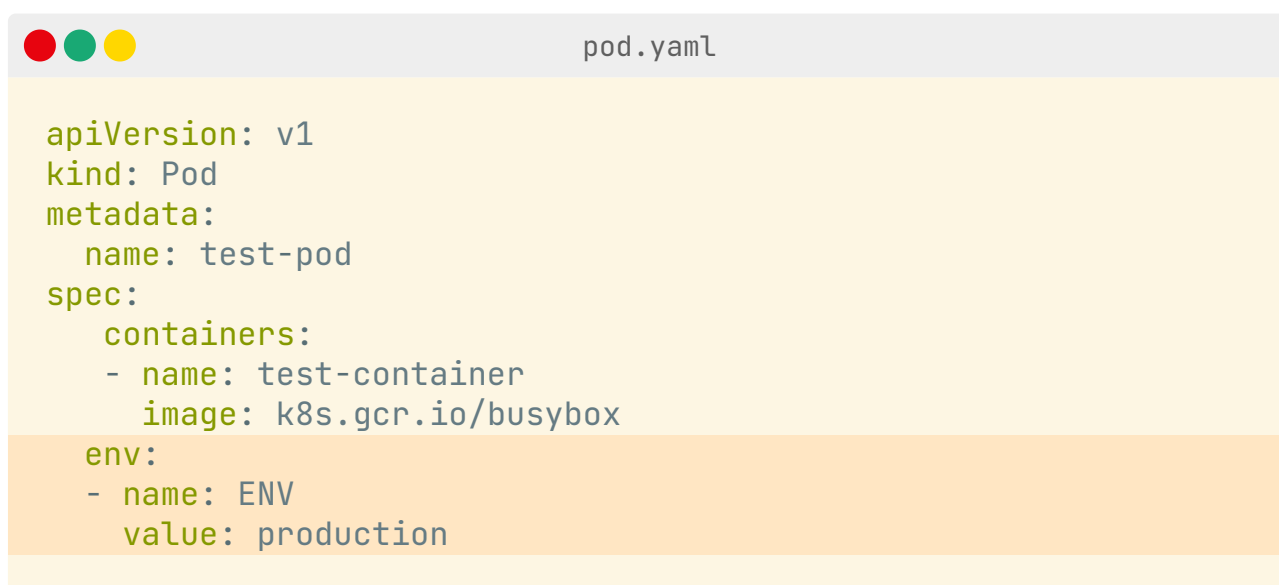
When you have multiple Kubernetes clusters, it's common to have YAML resources that can be applied to all environments but with a small modification.

As an example, you could have a Pod that expects an environment variable with the name of the environment.

Modern technologies such as Node.js and Ruby on Rails have specific variables to signal the current environment — `NODE_ENV` and `RAILS_ENV` respectively.

As an example, those variables are used not to load debugging tools when you're in production.

The following Pod is an example of what was just described:



```
apiVersion: v1
kind: Pod
metadata:
  name: test-pod
spec:
  containers:
  - name: test-container
    image: k8s.gcr.io/busybox
  env:
  - name: ENV
    value: production
```

Since the variable is hardcoded in the Pod, there's no easy way to deploy the same Pod in multiple environments.

However, you could create a Pod YAML definition for each of the environment you plan to

deploy.

Even if it is a temporary workaround, having several files has its challenges.

If you update the name of the image or the version, you should update all the files containing the same values.

Using templates with search and replace

A better strategy is to have a placeholder instead of the real value and replace it just before the resource is submitted to the cluster.

If you're familiar with `bash`, you can implement the search and replace with few lines of `sed`.


Your Pod should contain a placeholder like this:



```
apiVersion: v1
kind: Pod
metadata:
  name: test-pod
spec:
  containers:
  - name: test-container
    image: k8s.gcr.io/busybox
    env:
    - name: ENV
```

```
value: %ENV_NAME%
```

And you could run the following line to replace the value of the environment on the fly.

A terminal window with a title bar containing three colored circles (red, green, yellow) and the text "bash". The terminal content shows a command being executed: "\$ sed s/%ENV_NAME%/\$production/g \ pod_template.yaml > pod_production.yaml".

```
$ sed s/%ENV_NAME%/$production/g \
pod_template.yaml > pod_production.yaml
```

You could generate a YAML definition for any environment. You could also create the same Pod for dynamically created environments since you're not restricted to a fixed number of files.

But templating using `sed` has its limitations.

Perhaps you want to use a more friendly templating engine, or maybe you're after helper functions capable of converting the name of your variables into the snake-case format.

You could use your favourite templating engine and treat the YAML as a text file.

You could have placeholders and run your tool to create real YAML that you can submit to the cluster with `kubectl`.

You're not the only who had the same idea.

There're a lot of developers out there who had the same idea and came up with different competing templating solutions.

Here's a not comprehensive list of templating tools for

Kubernetes:

- kd
- terraform
- render
- kedge
- kdeploy
- ksonnet
- kmp
- OpenCompose
- kuku
- konfd
- Yipee.io
- forge
- quack
- mh
- kapitan
- kronjob
- ausroller
- kt
- incipit
- helm
- kustomize
- kontemplate
- bogie

- kubetpl
- kubeedn
- kmtpl
- kubor
- ntpl
- k8comp
- konfigenetes (my favourite name)

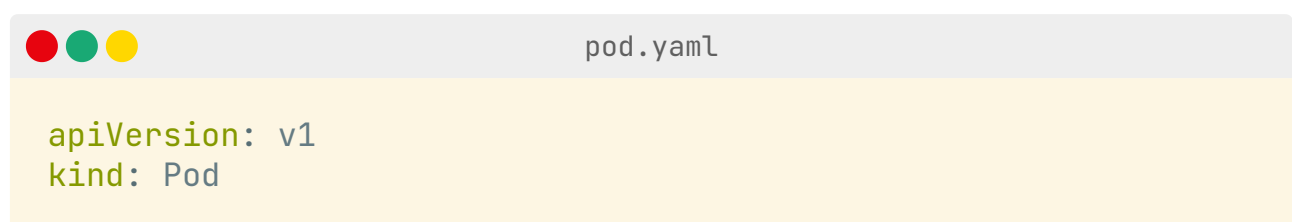
The list is long, should you write your own or use one from the list?

Helm — the templating engine

It turns out that there's a popular option and it's called Helm.

Helm can template YAML resources using the Golang templating engine and include a few helpers from the [Sprig library](#).

If you consider the example of the Pod it was presented earlier, this is what it looks like in Helm:



```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: test-pod
spec:
  containers:
  - name: test-container
    image: k8s.gcr.io/busybox
  env:
  - name: ENV
    value: {{ .Values.env_name }}
```

Please note that Helm interpolates values as a string and it doesn't understand YAML. You should take care of quotation, indentation etc. to form a valid manifest.

Helm is a binary that you install locally on your computer.

Charts: Helm's packages

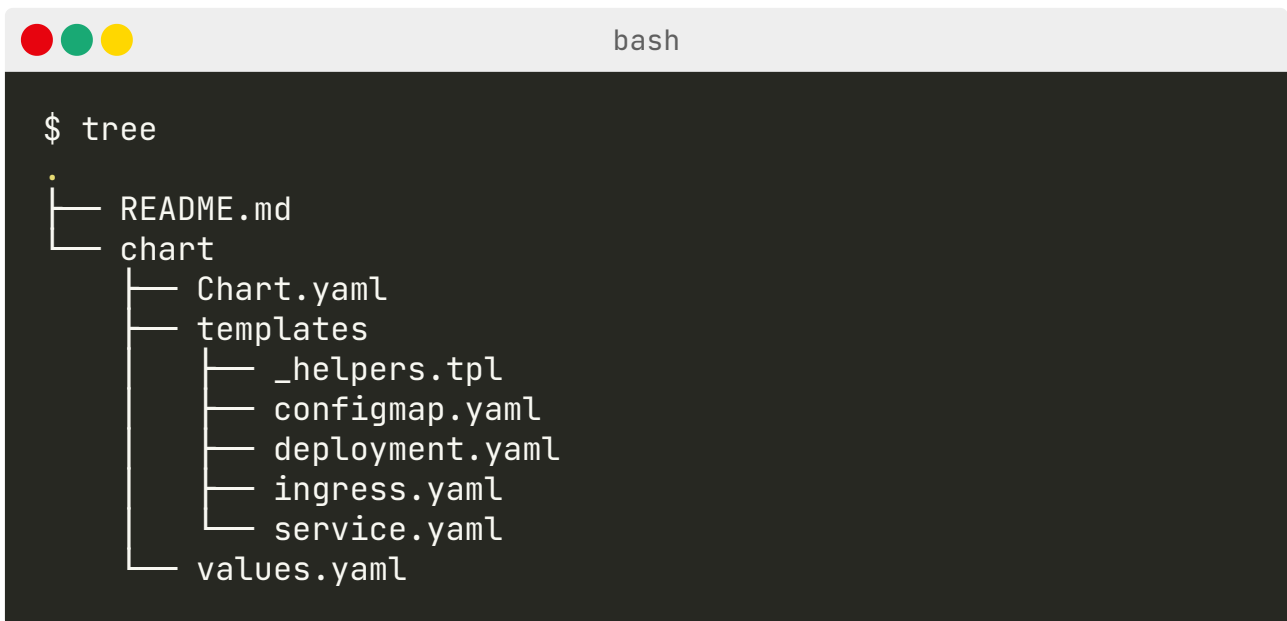
It's rare to deploy a single YAML at the time.

Most of the time, application are made of several components such as Deployment, Service, Ingress, ConfigMap, etc.

In Helm, you can group YAML resources into a single package called a Chart.

Helm Charts are a grouping of resources that follow a

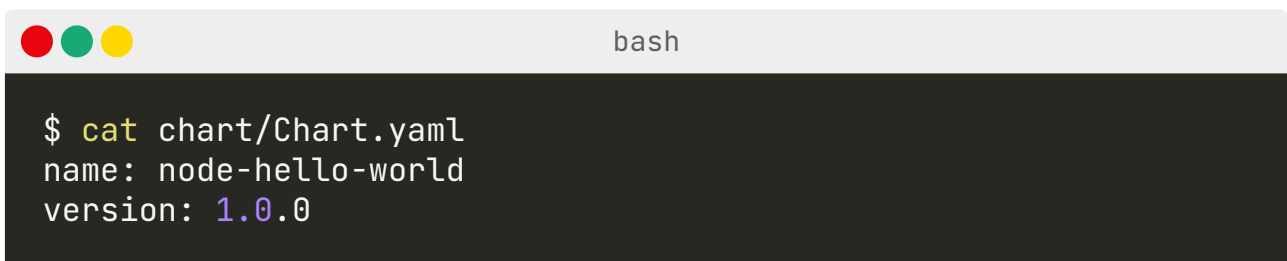
particular directory hierarchy.

A terminal window with a title bar containing three colored circles (red, green, yellow) and the text 'bash'. The terminal shows the command '\$ tree' and its output, which is a directory tree structure. The root directory contains 'README.md' and 'chart'. The 'chart' directory contains 'Chart.yaml', 'templates', and 'values.yaml'. The 'templates' directory contains '_helpers.tpl', 'configmap.yaml', 'deployment.yaml', 'ingress.yaml', and 'service.yaml'.

```
$ tree
.
├── README.md
└── chart
    ├── Chart.yaml
    ├── templates
    │   ├── _helpers.tpl
    │   ├── configmap.yaml
    │   ├── deployment.yaml
    │   ├── ingress.yaml
    │   └── service.yaml
    └── values.yaml
```

Let's start with the `Chart.yaml` file.

It's perhaps the most boring of all because it is designed to contain general information about your charts such as name and version.

A terminal window with a title bar containing three colored circles (red, green, yellow) and the text 'bash'. The terminal shows the command '\$ cat chart/Chart.yaml' and its output, which is the content of the 'Chart.yaml' file: 'name: node-hello-world' and 'version: 1.0.0'.

```
$ cat chart/Chart.yaml
name: node-hello-world
version: 1.0.0
```

A more exciting and noteworthy file is the `values.yaml`.

This file contains all of the variables that can be substituted in the templates.

With a glance, you can tell how you can customise the template.

Have a look at the `values.yaml` below:

A terminal window with a light gray title bar containing three colored circles (red, green, yellow) on the left and the text 'bash' in the center. The terminal has a dark background and displays the output of the command 'cat chart/values.yaml'. The output is: 'env_name: 'production'', 'image: learnk8s/node-hello-world', and 'replicas: 3'.

```
$ cat chart/values.yaml
env_name: 'production'
image: learnk8s/node-hello-world
replicas: 3
```

It's easy to imagine that you can customise the version of the container image used to deploy the app.

You can also personalise the replicas and the environment name.

As you probably already guessed, the chart is perhaps trying to create a reusable template for Deployments.

The templated resources are stored in the templates folder.

In that folder, you can have as many resources as you'd like.

Once the chart is installed in the cluster, all of YAML is rendered and submitted to the API.

That's excellent news: you don't have to `kubectl apply -f` for each YAML resource you created.

And even better, there's no typing `kubectl delete -f` to delete each resource.

You can delete the resources all at once with `helm delete`.

Helm — the release manager

The first time the chart is created, a new release is created.

A release is a collection of templates that were rendered and created from a group of resources.

You could think about the chart as Java classes and releases as instances of those classes.

From the same chart, you could have multiple releases, perhaps because you wish to have several templated copies of the same app.

A release is created automatically with every `helm install <release name> <chart folder>` command.

If you don't specify the name, a random name is automatically assigned to it as long you use the flag `--generate-name`.

The release name has to be unique, though.

A release has some useful bits of information associated with it, such as:

- The time that the release was created,
- The name of the release and
- The revision number associated

Please note that you can find the full list of details related to a release on [the official documentation](#)

The values above are available as fields in the prepopulated `.Release` variable, and you can use those in your template. Here's an example of how to do that:



```
apiVersion: v1
kind: Pod
metadata:
  name: |
    {{ printf "%s-%s" .Release.Name .Chart.Name }}
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      env:
        - name: ENV
          value: {{ .Values.env_name }}
```

The name of the Pod is namespaced with the name of the release.

So even if you have two Pods from the same chart, they are guaranteed to have a different release name.

Also, notice in this example how the `printf` helper was used to concatenate two values.

There're two useful commands in Helm that you should remember.

The first is listing releases with `helm ls`.

The command lists all current releases in the cluster, and it's convenient to keep track of things.

The other command is `helm upgrade <release name> <chart folder> .`

If you wish to amend your resources or tweak values in your `values.yaml`, you need to resubmit the chart.

But you don't want to create a release, so the `helm upgrade` command finds the right release and updates the resources associated with it.

When you have a lot of resources and several charts, you might find yourself repeating snippets of code frequently.

The templated Pod that features a concatenated release name and chart name is an excellent example of that.



```
pod.yaml

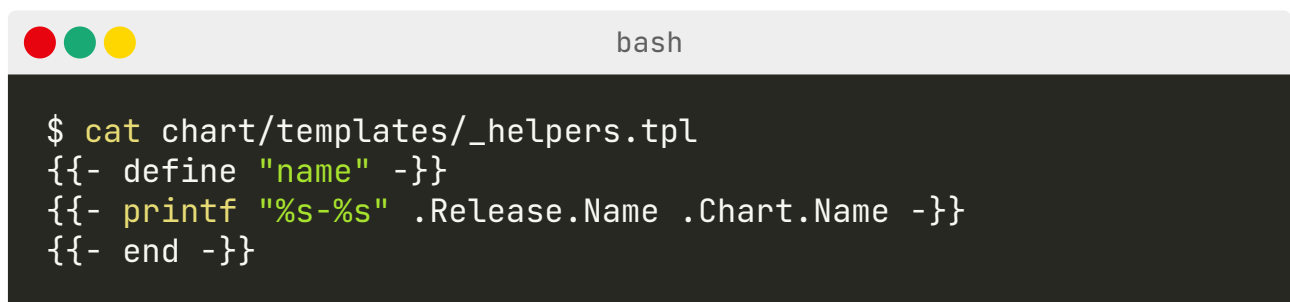
apiVersion: v1
kind: Pod
metadata:
  name: |
    {{ printf "%s-%s" .Release.Name .Chart.Name }}
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      env:
        - name: ENV
          value: {{ .Values.env_name }}
```

Organising snippet of code into templates

You can group frequently used code snippets into helpers and reuse them with shorthands.

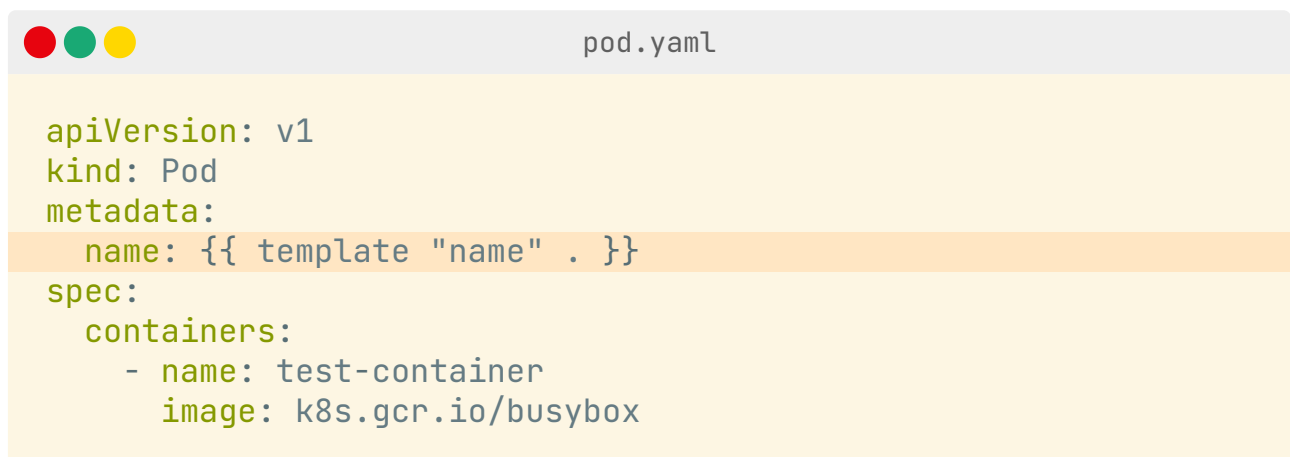
To create and helper you have to create a `_helpers.tpl` file in your `templates` folder and define the code that you wish to encapsulate.

Like this:

A terminal window with a title bar containing three colored circles (red, green, yellow) and the text 'bash'. The terminal shows the command to create a file and its contents.

```
$ cat chart/templates/_helpers.tpl
{{- define "name" -}}
{{- printf "%s-%s" .Release.Name .Chart.Name -}}
{{- end -}}
```

In your template, you can refer to the helper and avoid repeating yourself over and over again:

A YAML file editor window with a title bar containing three colored circles (red, green, yellow) and the text 'pod.yaml'. The file content is a Kubernetes Pod definition where a helper is used in the metadata name field.

```
apiVersion: v1
kind: Pod
metadata:
  name: {{ template "name" . }}
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
```



```
env:  
  - name: ENV  
    value: {{ .Values.env_name }}
```

Helpers are a great way to make your templates more idiomatic.

However, the Golang templating engine is quite primitive, and when you declare a helper, this is available globally.

That's something you should keep in mind when creating complex charts with nested dependencies such as the one discussed later on.

Managing releases and rollbacks in Helm

Every time you create or upgrade a release, the Helm holds a record of your requests into a table.

You might be familiar with this table already since it is the same table used by `kubectl` with the `--record` flag.

Still don't remember? You can head over to the Deployment Strategies section and read more about `kubectl` rollbacks.

The revision table is used as an auditing mechanism and more importantly, to roll back to a specific version when things go wrong.

1

| Revision | What happened |
|----------|---------------|
| | |

2

| Revision | What happened |
|----------|---------------------|
| 1 | Installed the chart |
| | |

3

| Revision | What happened |
|----------|---------------------|
| 1 | Installed the chart |
| 2 | Upgraded to v1.0.0 |
| | |

4

| Revision | What happened |
|----------|---------------------|
| 1 | Installed the chart |
| 2 | Upgraded to v1.0.0 |
| 3 | Upgraded to v2.0.0 |
| | |

Fig. 1 Helm records every resource that you submit.

Fig. 2 It records when the first deployment happened.

Fig. 3 It also records all the updates and upgrade you did to your releases.

Fig. 4 It also records all the updates and upgrade you did to your releases.

Helm has two useful commands to interact with versions:

- `helm history <release name>` — a convenient way to list all deployments for a particular release and
- `helm rollback <release name> <version>` — to roll back to a previous deployment.

Similarly to `kubectl rollback`, when you roll back to a previous deployment, the history is not rewritten.

Instead, a new row is inserted in the history table and the resources from that point in time are retrieved and submitted to the API.

1

| Revision | What happened |
|----------|---------------------|
| 1 | Installed the chart |
| 2 | Upgraded to v1.0.0 |
| 3 | Upgraded to v2.0.0 |
| | |

2

| Revision | What happened |
|----------|---------------------|
| 1 | Installed the chart |
| 2 | Upgraded to v1.0.0 |
| 3 | Upgraded to v2.0.0 |
| 4 | Rolled back to 2 |
| | |

Fig. 1 If you realise that a release contained a bug, you can rollback.

Fig. 2 Helm creates a new entry in the history table and reverts the resource definition to the previous deployment.

Why having `kubectl` rollbacks and Helm rollbacks? They both solve a similar problem.

Indeed, they do.

However, `kubectl` comes by default with Kubernetes, whereas Helm is an additional component that you might not want to use.

Helm isn't just a convenient way to template resources.

It can also package your YAML files in bundles called charts and is capable of managing releases.

That's probably why Helm is one of the most popular templating engines — at least for the time being.

It can do a lot more than templates.

But there's another reason why Helm is so famous, and it has to do with charts again.

Discovering and sharing charts

You learnt that a chart is just a collection of resources — YAML files.

Nothing stops you from zipping that folder and send it to your colleagues.

The sharing mechanism of charts further pushed the community to create public repositories of charts readily available to use in your cluster.

You can indeed type `helm search <name>` and find several charts with your favourite tool.

The community loved the idea of finding and sharing chart so much that made Helm even more widespread.

Declaring charts as dependencies

Charts are so popular that Helm supports dependencies between charts.

Imagine creating a Wordpress chart.

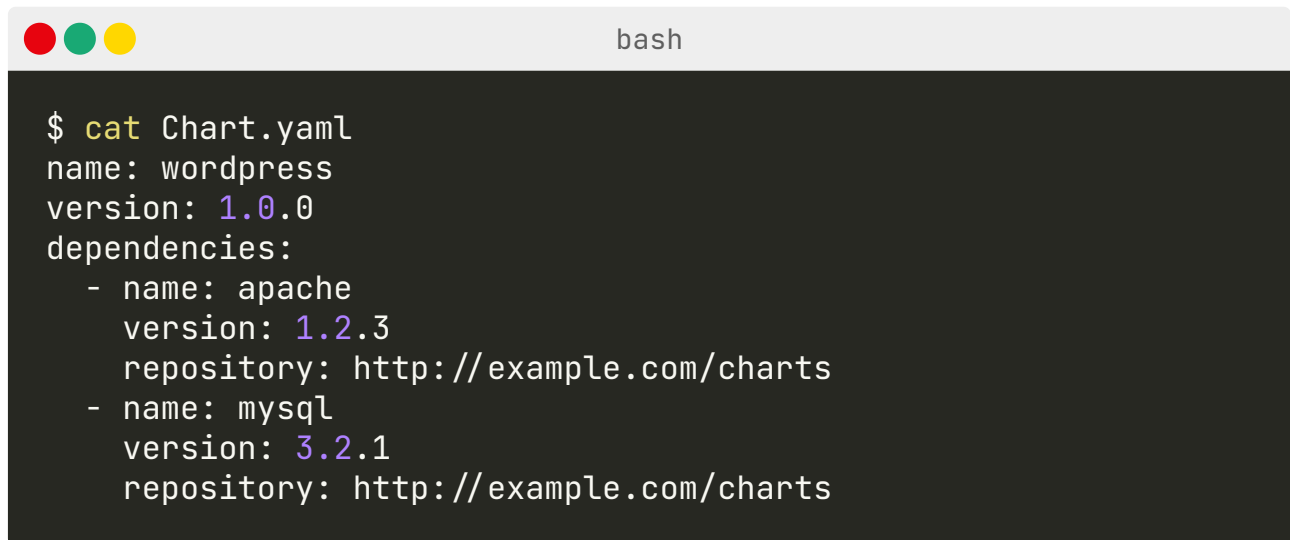
Wordpress is a blogging engine based on PHP and MySQL.

You could potentially create a Wordpress chart that defines

the PHP and MySQL chart as dependencies.

When you install it in your cluster, the dependencies are installed alongside your chart.

Dependencies are tracked into the file `Chart.yaml` and follow this structure:

A terminal window with a title bar containing three colored circles (red, green, yellow) and the text 'bash'. The terminal displays the output of the command 'cat Chart.yaml'.

```
$ cat Chart.yaml
name: wordpress
version: 1.0.0
dependencies:
  - name: apache
    version: 1.2.3
    repository: http://example.com/charts
  - name: mysql
    version: 3.2.1
    repository: http://example.com/charts
```

But why writing a Wordpress chart when there's one already available on the public registry?

Using private registries to share charts

If you're developing charts and wish to share them with the rest of the team privately, you can do so by uploading charts to a private registry.

Many existing registries already support Helm charts, but there's a favourite project that is gaining traction: [Chart Museum](#).

Helm 2 architecture

Helm 3 was a significant upgrade from the previous version and took years of development to complete.

Before learning how Helm3 works, it's worth understanding some of the decisions that led to today.

In the past, when you decided to deploy a resource, the template was sent not the Kubernetes API but to the Tiller — the Helm controller deployed in the cluster.

The Tiller was in charge of templating the resource and submitting it to the API.

Why not templating the resource client-side?

The tiller could query the Kubernetes API before submitting the resources.

So if there was any value that is dependent on a current Deployment, the Tiller could retrieve and inject it in the template.

And since the Tiller could interact with the Kubernetes API, it could also store the status of the resources you wish

to deploy.

Which it's handy, particularly when you make a mistake and want to roll back the change to the previous working version.

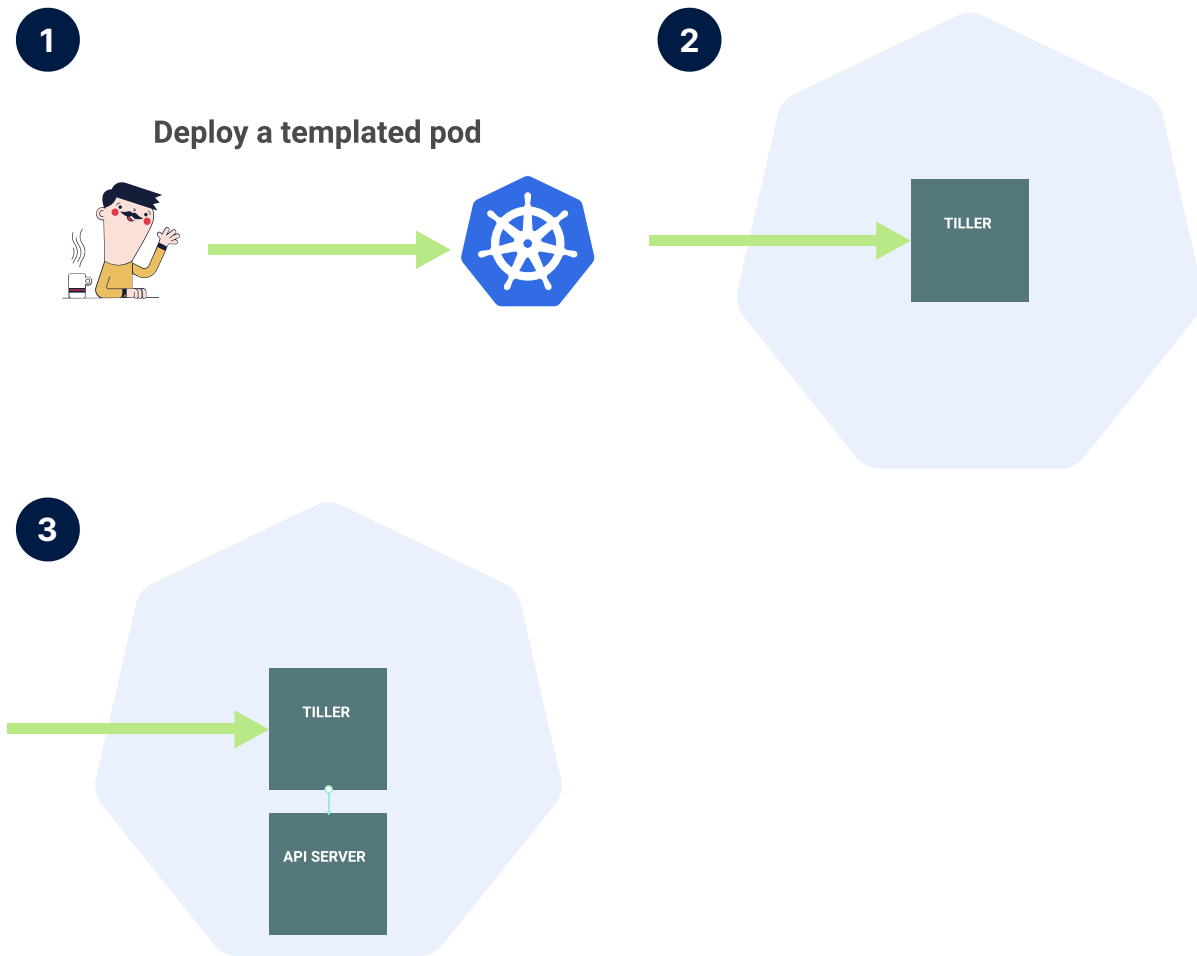


Fig. 1 When you create resources with Helm 2, the request was sent to your cluster.

Fig. 2 But before the resources was submitted to the API, they were intercepted and templated by an operator: the tiller.

Fig. 3 The tiller was in charge of storing previous resources used for deployments as well as templating the YAML.

Every time you send them a new release is created — unless you're upgrading an existing version.

The tiller tracked the resources and their associated releases. Having a controller templating resources in the cluster had its drawbacks, though.

It was another controller that you had to install, look after and secure.

Security was the primary concern when using Helm 2.

The Tiller could create any resource on your behalf, so it had to have full access to the API.

A malicious entity looking into attacking the cluster could leverage the Tiller to gain full access to it.

You could mitigate the risk of the Tiller being exploited by having it deployed on a per-namespace basis.

However, you still incurred the risk of the Tiller being compromised.

Helm 3 architecture

The Tiller was such a liability that the team behind Helm decided to remove in the next version: Helm 3.

If the Tiller was used to store information about releases, where are those stored in Helm 3?

Secrets.

Helm 3 is using Kubernetes Secrets to store details about releases.

It's also leveraging your existing credential (kubeconfig) when submitting resources to the cluster.

That means that there's that you can leverage existing toolchains and best practices.

Chapter 2

Installing Helm

Creating and deploying resources in your Kubernetes cluster is a complicated process.

If your application is made of a Java backend and a NodeJs frontend you probably have to create a Deployment and Service for each of them.

And don't forget an Ingress to route the traffic.

That's a total of five resources that need to be applied with `kubectl apply -f <file name> .`

Imagine you want to delete all of them. That's again five times `kubectl delete -f <file name> .`

But there's a better way to spend your time than typing the same thing five times.

The other annoyance comes from being able to deploy the code to multiple environments.

Indeed the code you deploy to staging or production may contain a specific Docker image instead of the latest build.

Would it be more convenient to have a systematic way to generate those yams files?

Perhaps something like a templating engine where you can specify your image as a variable that is substituted later on.

Enter Helm, the Kubernetes package manager.

Helm is used to creating deployments that can be templated. And it can do a lot more.

It's also possible to use Helm to upgrade, delete, inspect and rollback deployments.

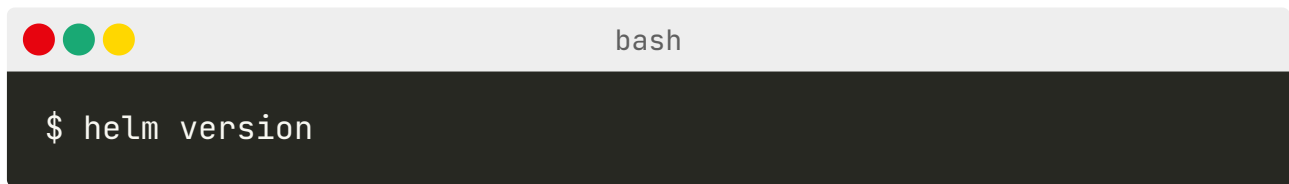
Prerequisites

You should have `minikube` installed with the Ingress addon enabled and `kubectl`.

Before you start playing with Helm, you need to install the binary.

[You can follow the instruction to install the client from here.](#)

If Helm was installed correctly, you should be able to run:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' in the center. The main area of the terminal is dark gray and shows a white prompt character '\$' followed by the text 'helm version'.

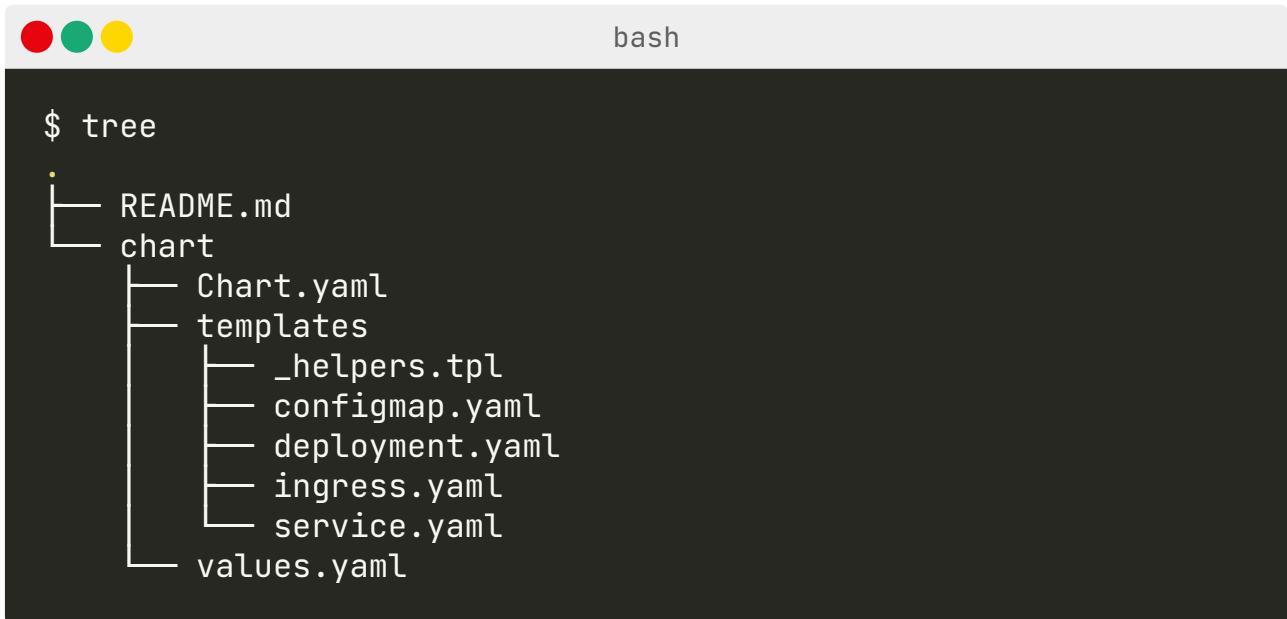
The version should be 3.0.0 or greater.

Chapter 3

Creating charts

Helm calls a chart a collection of resources such as a Deployment, a Service or an Ingress.

As a minimum, a chart has a `Chart.yaml` in the root folder and a folder called `templates` with all your resources.

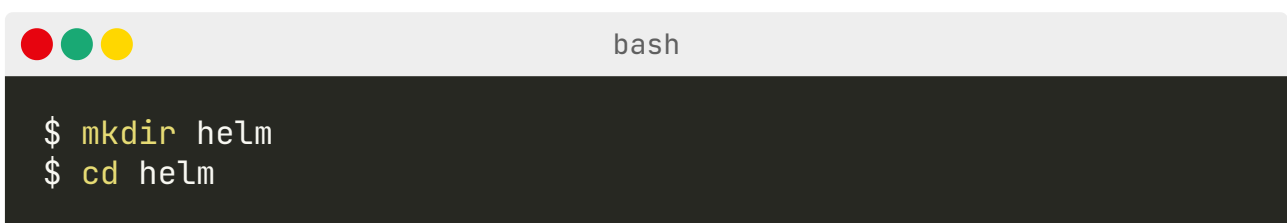


```
bash

$ tree
.
├── README.md
├── chart
│   ├── Chart.yaml
│   ├── templates
│   │   ├── _helpers.tpl
│   │   ├── configmap.yaml
│   │   ├── deployment.yaml
│   │   ├── ingress.yaml
│   │   └── service.yaml
│   └── values.yaml
```

Deploying the chart

You're going to deploy a simple *Hello World* application.
Create an empty directory:



```
bash

$ mkdir helm
$ cd helm
```

Create a `Chart.yaml` file with the following content:

```
Chart.yaml

name: node-app
version: 1.0.0
```

Create a folder name *templates*:

```
bash

$ mkdir templates
```

Inside the `templates` folder create a `deployment.yaml` file with the following content:

```
deployment.yaml

apiVersion: apps/v1
kind: Deployment
metadata:
  name: app
spec:
  replicas: 1
  selector:
    matchLabels:
      name: app
  template:
    metadata:
      labels:
        name: app
    spec:
      containers:
      - name: app
        image: learnk8s/hello:1.0.0
        imagePullPolicy: IfNotPresent
        ports:
```



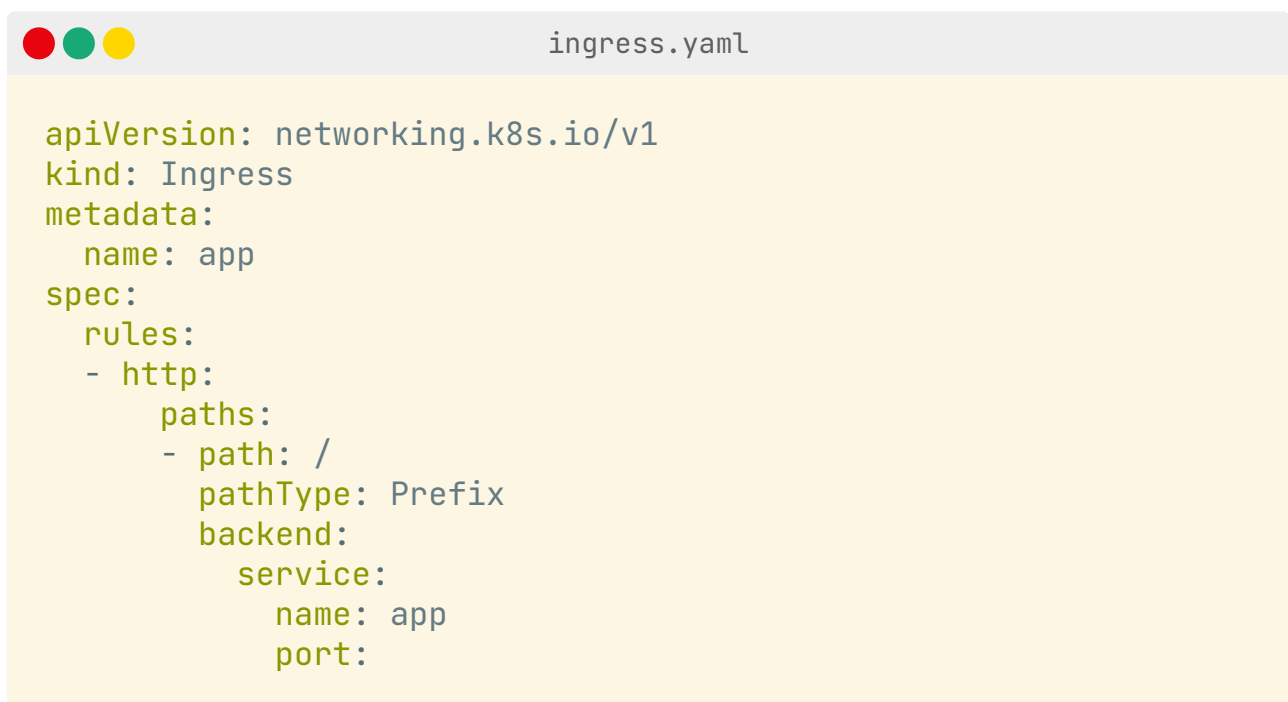
```
- containerPort: 3000
```

Also in the `templates` folder create a `service.yaml` file with the following content:



```
apiVersion: v1
kind: Service
metadata:
  name: app
spec:
  ports:
    - port: 80
      targetPort: 3000
  selector:
    name: app
```

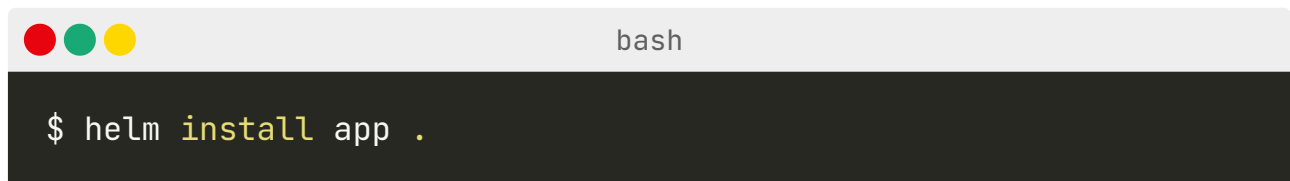
Also in the `templates` folder create an `ingress.yaml` file with the following content:



```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app
spec:
  rules:
    - http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: app
                port:
```

```
number: 80
```

You can deploy the chart (and all the resources you created) with the following command:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ helm install app .' in a light-colored monospace font.

```
$ helm install app .
```

Where `.` is the folder containing your chart (i.e. the current folder).

Helm should have applied all your resources at once.

No need to create resources one by one.

Helm also created a release.

You can list all active releases with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ helm ls' in a light-colored monospace font.

```
$ helm ls
```

Testing the application

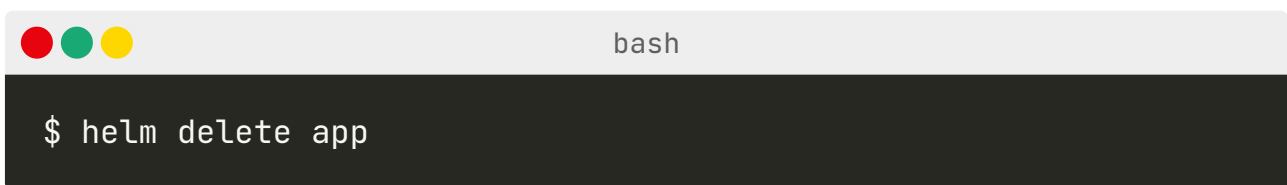
Retrieve the minikube IP to visit the application:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' in the center. The main area of the terminal is dark gray and contains the command '\$ minikube ip' in a light-colored monospaced font.

```
bash
$ minikube ip
```

The application is available at `http://<replace with minikube ip>/`

You can delete a release with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' in the center. The main area of the terminal is dark gray and contains the command '\$ helm delete app' in a light-colored monospaced font.

```
bash
$ helm delete app
```

The delete command lets you delete all the resources at once.

There's no need to delete resources one by one manually.

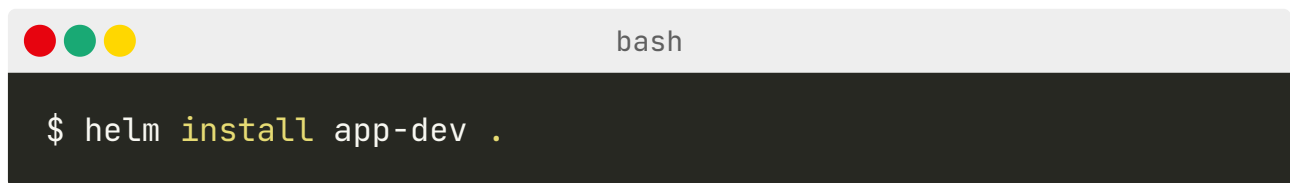
Chapter 4

Templating resources

What if you want to deploy the same application twice?

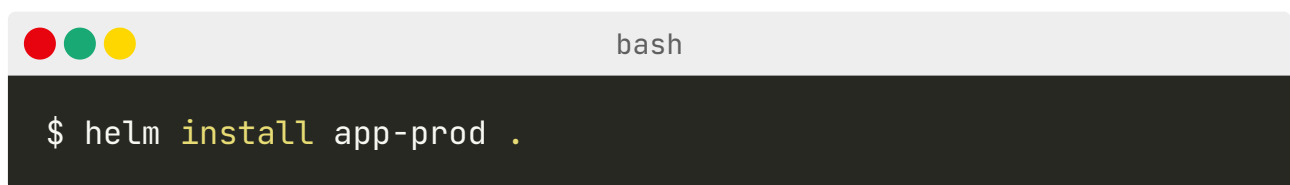
Perhaps you have a development and production environment.

You could deploy the first environment with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ helm install app-dev .' in a light-colored monospaced font.

```
$ helm install app-dev .
```

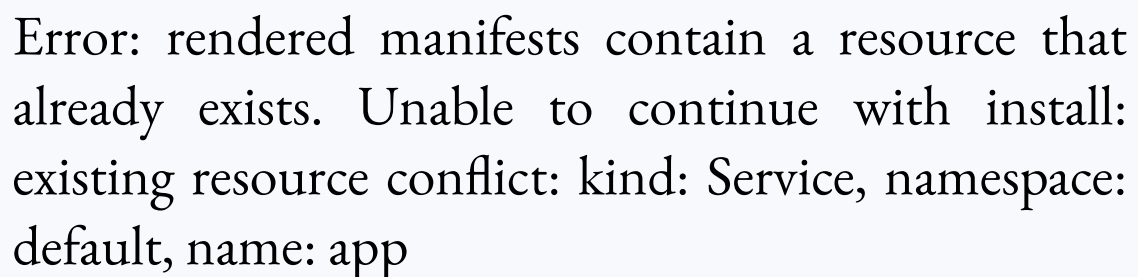
And repeat the command for a second environment with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ helm install app-prod .' in a light-colored monospaced font.

```
$ helm install app-prod .
```

But it doesn't work.

You should have noticed the following error:

A light blue rectangular box with a thin blue vertical line on its left side. It contains an error message in a black monospaced font.

```
Error: rendered manifests contain a resource that  
already exists. Unable to continue with install:  
existing resource conflict: kind: Service, namespace:  
default, name: app
```

As you may have already guessed, you can't have two apps with the same resource names in the same namespace.

Templated deployments

But the two releases have different names, so it'd be nice if you could include the name of the release to in your resource definition.

Helm lets you create templated resources.

Perhaps you could prefix the release name in front of the name of the resource.

You could use the release name that is exposed as a variable called `.Release.Name`.

You can update the `deployment.yaml` file to include it like so:

```
deployment.yaml

apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ printf "%s-%s" .Release.Name .Chart.Name }}
spec:
  replicas: 1
  selector:
    matchLabels:
      name: {{ printf "%s-%s" .Release.Name .Chart.Name }}
  template:
    metadata:
      labels:
        name: {{ printf "%s-%s" .Release.Name .Chart.Name }}
    spec:
      containers:
        - name: app
          image: learnk8s/hello:1.0.0
```

```
imagePullPolicy: IfNotPresent
ports:
  - containerPort: 3000
```

Notice how values are templated using a double curly brace. Helm Chart templates are written in the [Go template language](#), with the addition of 50 or so add-on template functions from [the Sprig library](#).

There are many predefined values like:

- `.Release` is used to refer to the resulting release values, e.g. `.Release.Name` , `.Release.Time` .
- `.Chart` is used to refer to the `Chart.yaml` configuration.
- `.Files` is used to refer to the files in the chart directory.

You should update the `service.yaml` file too:



```
service.yaml

apiVersion: v1
kind: Service
metadata:
  name: {{ printf "%s-%s" .Release.Name .Chart.Name }}
spec:
  ports:
    - port: 80
      targetPort: 3000
  selector:
    name: {{ printf "%s-%s" .Release.Name .Chart.Name }}
```

And also update the `ingress.yaml` file:

```
ingress.yaml

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: {{ printf "%s-%s" .Release.Name .Chart.Name }}
spec:
  rules:
  - host: {{ .Release.Name }}.hello-world.dev
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: {{ printf "%s-%s" .Release.Name .Chart.Name }}
            port:
              number: 80
```

You can now deploy your application how many time you want to:

```
bash

$ helm install dev .
$ helm install prod .
```

You can list Pods, Services and Ingresses with:

```
bash

$ kubectl get pods,services,ingress
```

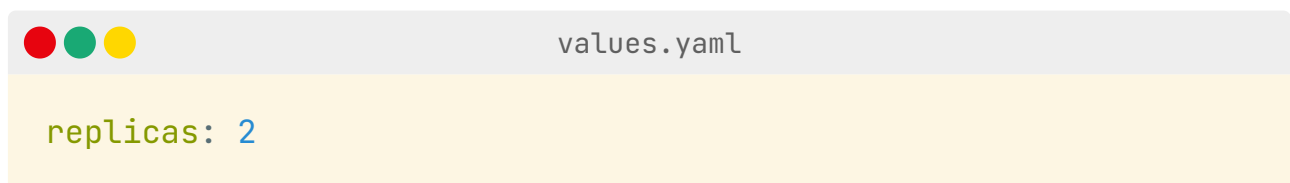
You have installed multiple versions of your applications.

Defining your variables

There's also a way to define your variables.

Custom values for the templates are supplied using a `values.yaml`.

Create a `values.yaml` file in the root directory with the following content:

A screenshot of a code editor window titled 'values.yaml'. The editor has a light yellow background and a grey title bar with three colored window control buttons (red, green, yellow) on the left. The code inside is 'replicas: 2' in a monospaced font, with 'replicas:' in green and '2' in blue.

```
replicas: 2
```

Values that are supplied via a `values.yaml` file are accessible from the `.Values` object in a template.

You can parametrise the replicas like so in your `deployment.yaml` file:

A screenshot of a code editor window titled 'deployment.yaml'. The editor has a light yellow background and a grey title bar with three colored window control buttons (red, green, yellow) on the left. The code inside is a Kubernetes deployment manifest. The line 'replicas: {{ .Values.replicas }}' is highlighted with an orange background.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ printf "%s-%s" .Release.Name .Chart.Name }}
spec:
  replicas: {{ .Values.replicas }}
  selector:
    matchLabels:
      name: {{ printf "%s-%s" .Release.Name .Chart.Name }}
  template:
    metadata:
```

```
labels:
  name: {{ printf "%s-%s" .Release.Name .Chart.Name }}
spec:
  containers:
  - name: app
    image: learnk8s/hello:1.0.0
    imagePullPolicy: IfNotPresent
    ports:
      - containerPort: 3000
```

You can install the application with:

A terminal window with a light gray title bar containing three colored circles (red, green, yellow) on the left and the text 'bash' on the right. The terminal area is dark gray and shows the command '\$ helm install staging .' in white text.

```
bash
$ helm install staging .
```

You should check that the new release has two pods with:


A terminal window with a light gray title bar containing three colored circles (red, green, yellow) on the left and the text 'bash' on the right. The terminal area is dark gray and shows the command '\$ kubectl get pods' in white text.

```
bash
$ kubectl get pods
```

Overwriting custom values

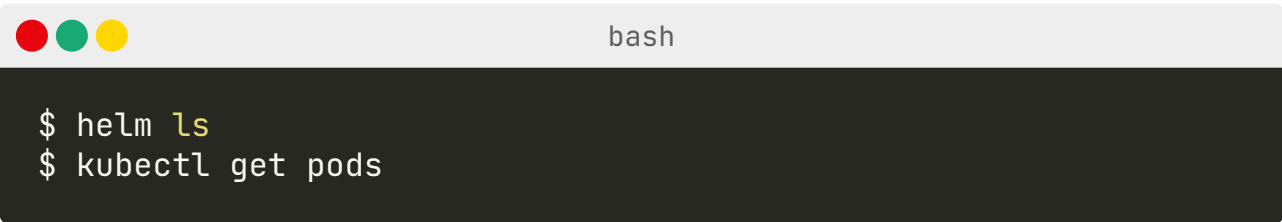
Values in the `values.yaml` can be overwritten at the time of making the release using `--values YAML_FILE_PATH` or `--set key1=value1,key2=value2` parameters.

You could change the replicas with:



```
bash
$ helm install preprod . --set replicas=3
```

One more release was created with three Pods.
You can verify the release with:



```
bash
$ helm ls
$ kubectl get pods
```

Before deploying the application, it's useful to check that the files are generated correctly.

Dry-run

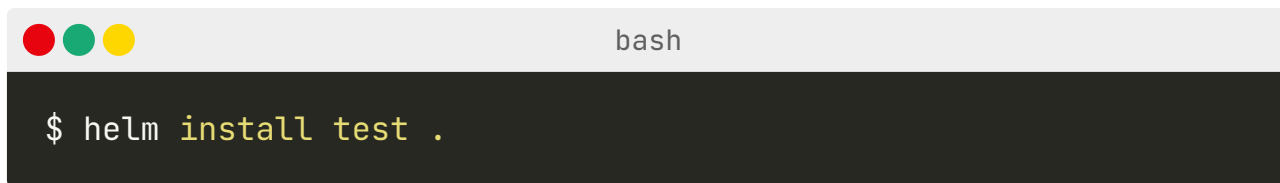
You can perform a dry-run with:



```
bash
$ helm install test . --dry-run --debug
```

Where `.` is the folder containing your chart.

If you're happy with the result, you can go ahead and deploy it again with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ helm install test .' in a light-colored monospaced font.

```
bash
$ helm install test .
```

Managing releases

You can list all active releases with:

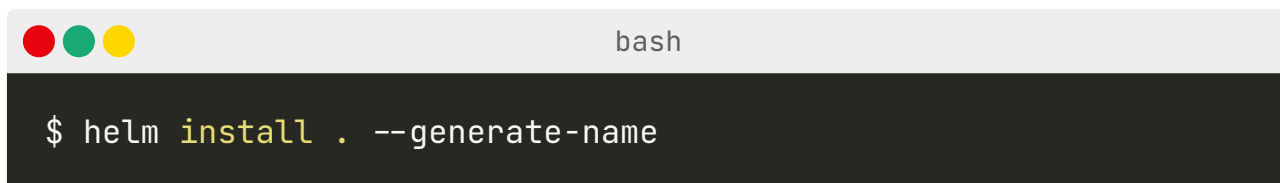
A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ helm ls' in a light-colored monospaced font.

```
bash
$ helm ls
```

You can delete a release with `helm delete <release-name> .`
Go ahead and delete all existing releases.

In the previous examples, you always picked a name for your release.

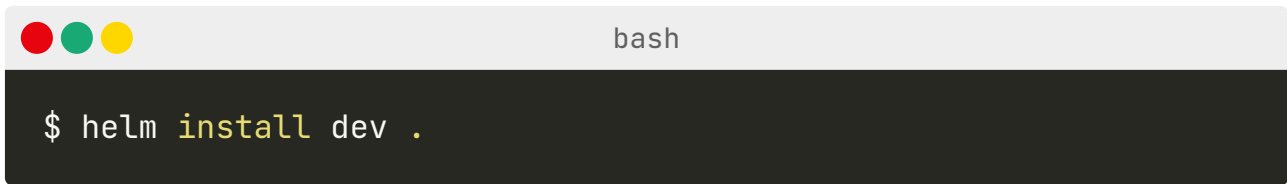
You can let Helm pick the release name for you with the flag `--generate-name` like so:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The main area of the terminal is dark gray and contains the command '\$ helm install . --generate-name' in a light-colored monospaced font.

```
bash
$ helm install . --generate-name
```

It's usually more convenient to have a predictable naming strategy, though.

Let's create a release with the name *dev*:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text "bash" in the center. The main area of the terminal is dark gray and contains the command "\$ helm install dev ." in a light green monospaced font.

```
bash
$ helm install dev .
```

The application is available at dev.hello-world.dev:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text "bash" in the center. The main area of the terminal is dark gray and contains the command "\$ curl --header 'Host: dev.hello-world.dev' \$(minikube ip)" in a light green monospaced font.

```
bash
$ curl --header 'Host: dev.hello-world.dev' $(minikube ip)
```

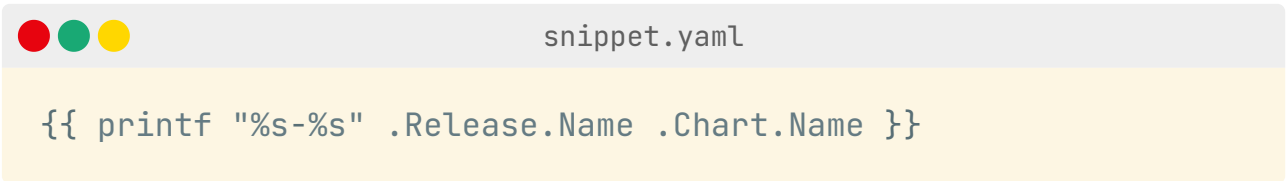
Of course, you could do the same with an autogenerated name.

However, naming releases could help you manage your environments as you practice continuous delivery.

Chapter 5

Helpers

You have probably noticed a repeating pattern in the templates:



```
{{ printf "%s-%s" .Release.Name .Chart.Name }}
```

This repetition becomes even more apparent in large applications, with dozens of different resources.

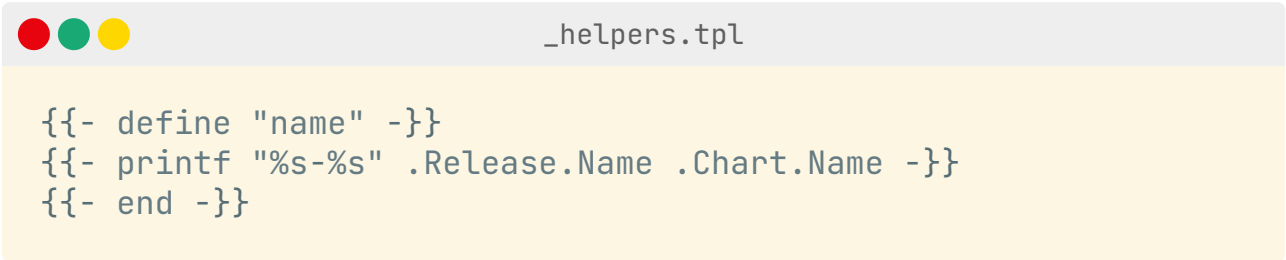
But you can avoid repeating yourself over and over with partials.

Partials

Partials are snippets of code that can be embedded and expanded in other templates.

Since those files are not rendered as regular templates, they're prefixed with an underscore.

Let's create a `_helpers.tpl` in the `templates` folder with the following content:



```
{{- define "name" -}}  
{{- printf "%s-%s" .Release.Name .Chart.Name -}}  
{{- end -}}
```

There is one crucial detail to keep in mind when naming templates: template names are global.

If you declare two templates with the same name, whichever one is loaded last will be the one used.

Because templates in sub charts are compiled together with top-level templates, you should be careful to name your templates with chart-specific names.

One popular naming convention is to prefix each defined template with the name of the chart: `{{ define "mychart.name" }}` or `{{ define "mychart_name" }}`.

You can now update all the resources starting from the `deployment.yaml` file:



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ template "name" . }}
spec:
  replicas: {{ .Values.replicas }}
  selector:
    matchLabels:
      name: {{ template "name" . }}
  template:
    metadata:
      labels:
        name: {{ template "name" . }}
    spec:
      containers:
        - name: app
          image: learnk8s/hello:1.0.0
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 3000
```


To render the template, you have to call the `template` function and provide two arguments: the name of the helper and the scope.

The latter is used to read the parent values such as `.Release.Name` and `.Chart.Name`.

You can update the `service.yaml` file too:

```
service.yaml

apiVersion: v1
kind: Service
metadata:
  name: {{ template "name" . }}
spec:
  ports:
    - port: 80
      targetPort: 3000
  selector:
    name: {{ template "name" . }}
```

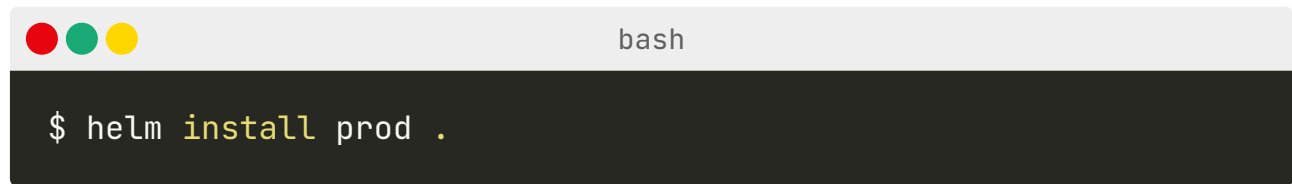
And the `ingress.yaml` :

```
ingress.yaml

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: {{ template "name" . }}
spec:
  rules:
    - host: {{ .Release.Name }}.hello-world.dev
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
```


```
name: {{ template "name" . }}  
port:  
  number: 80
```

You can install the chart with:



```
bash  
$ helm install prod .
```

You can verify that the chart and the app still work with:



```
bash  
$ curl --header 'Host: prod.hello-world.dev' $(minikube ip)
```

Chapter 6

Updating charts

Let's assume you need to update a label on the Deployment. You could edit the Deployment resource directly in Kubernetes with `kubectl edit`, but it is inconvenient. In fact, you have to edit all of the Deployments, one for each environment.

Upgrading releases

A better strategy is to change the label once and let Helm do the work of updating and versioning releases.

Let's start by changing your `deployment.yaml` file:



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ template "name" . }}
spec:
  replicas: {{ .Values.replicas }}
  selector:
    matchLabels:
      name: {{ template "name" . }}
  template:
    metadata:
      labels:
        name: {{ template "name" . }}
        track: canary
    spec:
      containers:
      - name: app
        image: learnk8s/hello:1.0.0
```

```
imagePullPolicy: IfNotPresent
ports:
  - containerPort: 3000
```

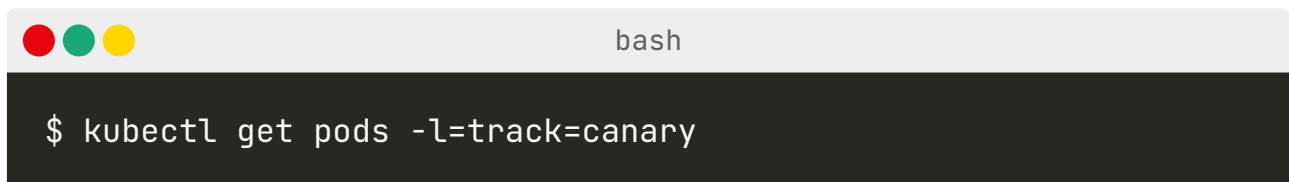
You can update the development release with:



```
bash
$ helm upgrade prod .
```

Where *prod* is the name of the release and `.` the folder containing the chart.

You can verify that the label was applied correctly with:

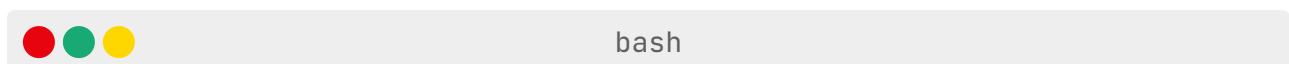


```
bash
$ kubectl get pods -l=track=canary
```

Rolling back releases

Helm deployed the application and versioned the deployment.

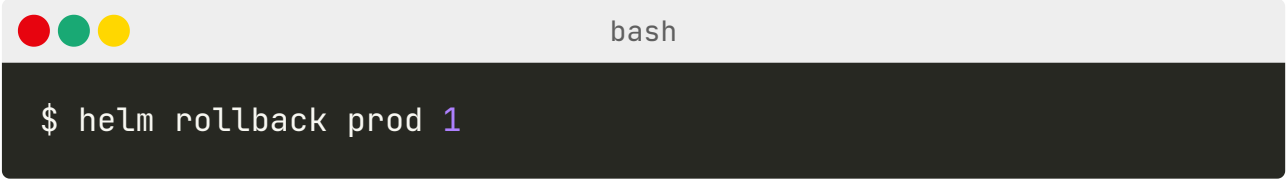
You can check the history of your deployments with:



```
bash
```

```
$ helm history prod
```

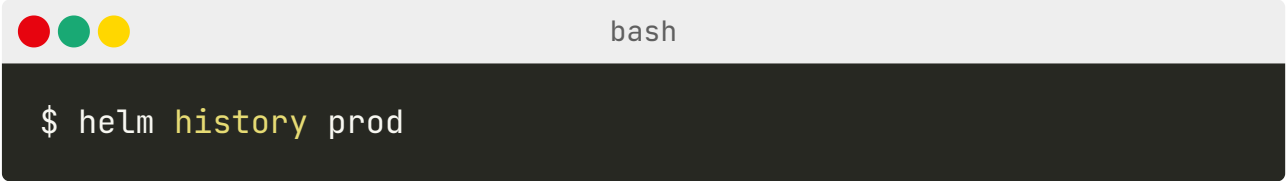
If you want to revert the release to a previous point in time, you can do so with:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The terminal area has a dark background and shows the command '\$ helm rollback prod 1' in a light-colored font.

```
$ helm rollback prod 1
```


Where *prod* is the release name and *1* is the version you wish to rollback.

You can verify the rollback was successful by inspecting the history:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The terminal area has a dark background and shows the command '\$ helm history prod' in a light-colored font.

```
$ helm history prod
```

Since the first release didn't have the `track: canary` label, you should expect zero pods from the following command:

A terminal window with a light gray title bar containing three colored window control buttons (red, green, yellow) on the left and the text 'bash' on the right. The terminal area has a dark background and shows the command '\$ kubectl get pods -l=track=canary' in a light-colored font.

```
$ kubectl get pods -l=track=canary
```